

bookboon.com

Hugh Darwen

SQL: A Comparative Survey

SQL: A Comparative Survey

2th edition

© 2014 Hugh Darwen & bookboon.com

ISBN 978-87-403-0778-8

Contents

Preface	9
1 Introduction	11
1.1 Introduction	11
1.5 “Collection of Variables”	13
1.6 What Is an SQL Database?	14
1.7 “Table” Not Equal to “Relation”	15
1.8 Anatomy of a Table	16
1.9 What Is a DBMS?	17
1.10 SQL Is a Database Language	17
1.11 What Does an SQL DBMS Do?	18
1.12 Creating and Destroying Base Tables	18
1.13 Taking Note of Integrity Rules	20
1.14 Taking Note of Authorisations	21
1.15 Updating Variables	22
1.16 Providing Results of Queries	25



Strømmen produseres ofte langt fra der den skal brukes.

Statnett sitt oppdrag er å gjøre strømmen tilgjengelig, uansett hvor i dette langstrakte landet du bor. Det er vi som bygger og drifter "riksveiene" i norsk strømforsyning. Gjennom vårt landsdekkende nett sørger vi for en sikker fordeling av strøm mellom nord, sør, øst og vest.

Vi binder Norge sammen

Statnett
Vårt felles kraftnett

Er du student? Les mer her
www.statnett.no/no/Jobb-og-karriere/Studenter



2	Values, Types, Variables, Operators	26
2.1	Introduction	26
2.2	Anatomy of A Command	29
2.3	Important Distinctions	30
2.4	A Closer Look at a Read-Only Operator (+)	30
2.5	Read-only Operators in SQL	30
2.6	What Is a Type?	36
2.7	What Is a Type Used For?	39
2.8	The Type of a Table	40
2.9	Table Literals	42
2.10	Types and Representations	44
2.11	What Is a Variable?	50
2.12	Updating a Variable	55
2.13	Conclusion	57
3	Predicates and Propositions	59
3.1	Introduction	59
3.2	What Is a Predicate?	60
3.3	Substitution and Instantiation	61
3.4	How a Table Represents an Extension...	62
3.5	Deriving Predicates from Predicates	62

Hva får egentlig en ingeniør- eller teknologistudent for 300 kroner?

- Medlemskap i en aktiv studentorganisasjon – hele studietiden
- 150 tillitsvalgte studenter som ivaretar dine interesser
- Jobbsøkerkurs
- Gratis PC-forsikring og gode bank- og forsikringstilbud
- Teknisk Ukeblad og NITO Refleks
- Møteplasser på web 2.0

Flere medlemsfordeler og innmelding: www.nito.no/student

Alle som studerer på ingeniør-, bioingeniør-, sivilingeniør eller andre teknologistudier (høgskolekandidat, bachelor eller master) kan bli medlem i NITO.

NITO NORGES STØRSTE ORGANISASJON
FOR INGENIØRER OG TEKNOLOGER



4	Relational Algebra— The Foundation	73
4.1	Introduction	73
4.2	Relations and Predicates	81
4.3	Relational Operators and Logical Operators	81
4.4	JOIN and AND	82
4.5	Renaming Columns	86
4.6	Projection and Existential Quantification	88
4.7	Restriction and AND	92
4.8	Extension and AND	94
4.9	UNION and OR	96
4.10	Semidifference and NOT	100
4.11	Concluding Remarks	104
	EXERCISES	105
5	Building on The Foundation	107
5.1	Introduction	107
5.2	Semijoin and Composition	108
5.3	Aggregate Operators	114
5.4	Tables within a Table	118



Skatteetaten



Vil du jobbe i et av landets største IT-miljøer?
Vi skal gjøre det kompliserte enkelt

Skatteetaten tilbyr store fagmiljø og utfordrende oppgaver innen:

- > Systemutvikling
- > Service oriented architecture (SOA)
- > Business intelligence (BI)
- > Testledelse
- > Webutvikling
- > IT sikkerhet
- > Infrastruktur
- > Brukergrensesnitt

For nyutdannede IT-spesialister kan vi tilby et to-årig traineeprogram.

For mer informasjon se skatteetaten.no/jobb

Profesjonell • Nytenkende • Imøtekommende



5.5	Using Aggregation on Nested Tables	120
5.6	Summarization in SQL	124
5.7	Grouping and Ungrouping in SQL	126
5.8	Wrapping and unwrapping in SQL	131
5.9	Table Comparison	132
5.10	Other Operators on Tables and Rows	133
	EXERCISES	134
6	Constraints and Updating	136
6.1	Introduction	136
6.2	A Closer Look at Constraints and Consistency	140
6.3	Expressing Constraint Conditions	141
6.4	Useful Shorthands for Expressing Some Constraints	148
6.5	Updating Tables	155
	EXERCISES	164
	Appendix A: References and Bibliography	167



OLJE- OG ENERGIDEPARTEMENTET



Er du full av energi?

Olje- og energidepartementets hovedoppgave er å tilrettelegge for en samordnet og helhetlig energipolitikk. Vårt overordnede mål er å sikre høy verdiskapning gjennom effektiv og miljøvennlig forvaltning av energiressursene.

Vi vet at den viktigste kilden til læring etter studiene er arbeidssituasjonen. Hos oss får du:

- Innsikt i olje- og energisektoren og dens økende betydning for norsk økonomi
- Utforme fremtidens energipolitikk
- Se det politiske systemet fra innsiden
- Høy kompetanse på et saksfelt, men også et unikt overblikk over den generelle samfunnsutviklingen
- Raskt ansvar for store og utfordrende oppgaver
- Mulighet til å arbeide med internasjonale spørsmål i en næring der Norge er en betydelig aktør

Vi rekrutterer sivil- og samfunnsøkonomer, jurister og samfunnsvitere fra universiteter og høyskoler.

www.regjeringen.no/oed



Se ledige stillinger her

www.jobb.dep.no/oed



This book is dedicated to members past and present of ISO/IEC Joint Technical Committee 1, Subcommittee 32, Working Group 3, “Database Languages”.

Preface

This book is a companion to my *An Introduction to Relational Database Theory* (reference [7]) and is intended to be read in parallel with that text, hereinafter referred to as “the theory book”. As I noted in the preface of that book, a university course on relational databases is typically divided into a theory component and an “industrial” component requiring the student to learn the basics of SQL. In that preface I also mention that in my own teaching I encourage students to “compare and contrast SQL with what they have learned in the theory part”. This book is my own attempt to compare and contrast it, piece by piece, with what I have taught in the theory book.

The structure of the book closely parallels the first six chapters of the theory book, and the structure of each chapter is such that each section, example, and figure corresponds by number to its counterpart in the theory book. There are some gaps in the numbering. These arise when there is something in the theory book for which there is nothing relevant to discuss in the context of SQL. Conversely, I sometimes need several SQL examples in connection with a single example in the theory book, in which case I suffix the example numbers with a, b, c, and so on.

I do not include counterparts of the last two chapters of the theory book. That’s because they deal with relational database design issues and the treatment, insofar as it goes, is equally applicable to SQL databases.

Although existing knowledge of SQL is not a prerequisite, this book is not intended to be used as an SQL primer. Rather, its aim is to investigate the extent to which SQL supports and adheres to the theory, how it does so, and where and how it departs from the theory. Inevitably, the book also provides opportunities for comment on SQL’s degree of adherence to commonly accepted principles of computer language design.

Much of the study involves comparing expressions in **Tutorial D** with equivalent or near-equivalent expressions in SQL. Being one of the two inventors of **Tutorial D** (the other is Chris Date), I am obviously exposing myself to possible accusations of deliberately choosing examples that show my own language in a favourable light. Let me just say, in advance of any such accusations, that I had no idea of writing this book when I chose the examples for the theory book, and I have tried hard to avoid over-elaborate or otherwise perverse SQL formulations. I am open to suggestions and willing to make revisions if simplifications of any of my examples come to light.

I stick to *standard* SQL (ISO/IEC 9075—see reference [15]) in all my examples. As a member of the relevant committee (currently ISO/IEC JTC 1/SC 32/WG 3) from 1989 to 2004 I was deeply involved in the drafting of the various documents constituting that standard. I have taken advantage of that experience by sprinkling my text with subsections headed **Historical Notes**, these being clearly demarcated in **brown** to help you skip them if they don't interest you. SQL, whose first commercial implementation appeared in 1979, has grown out of all proportion to its original self over the years and the main aim of these historical notes is to tell you when and why the various features that I use in my examples arose in the language. I was not involved in IBM's original development of SQL, so there is a certain amount of conjecture—albeit reasonably well-informed conjecture—where I write about the early history of the language. Various SQL guides based on the standard are mentioned in Appendix A (references [2], [10], and [16]). These were all published in the 1990s and thus do not cover material that was added in SQL:1999 and subsequent editions.

I am well known in some quarters as a severe critic of SQL. I have tried hard not to burden readers with my personal opinions but, rather, just to present the facts and let readers form their own opinions. In spite of my efforts I have possibly not been 100% successful in that endeavour. It was difficult.

Acknowledgments

My initial draft was reviewed by Jamie Collins, Chris Date, and Erwin Smout, all of whom made some useful comments. Needless to say, all the remaining errors—and there are sure to be some—are my own.

Karin Hamilton Jakobsen, my contact at Ventus Publishing, has been very helpful in many respects in connection with all my books published by that company.

In spite of my declared distaste for the language, SQL, I cannot speak too highly of my fellow members of the aforementioned standards committee, especially Jim Melton, the editor-in-chief since 1987, successive convenors Len Gallagher, Stephen Cannan and Keith Hare, and my former colleagues in the UK delegation, Ed Dee, Mike Sykes and Phil Brown. Of course there are many others, from many national bodies and far too numerous to be named here, who have also showed great dedication and professionalism in the monumental task of producing this standard. My appreciation is expressed in my dedication for this book.

1 Introduction

1.1 Introduction

Chapter 1 of the theory book gives a very broad overview of

- what a database is
- what a relational database is, in particular
- what a database management system (DBMS) is
- what a DBMS does
- how a relational DBMS does what a DBMS does

It introduces you to the terminology and notation used in the remainder of the book and gives a brief introduction to each topic that is covered in more detail in subsequent sections. The first, third and fourth bullet points clearly apply equally well in the SQL world and I have nothing to add on those topics here. However, the second and last bullet points need to be replaced, respectively, by

- what an SQL database is, in particular

and

- how an SQL DBMS does what a DBMS does

Sections 1.2 to 1.5 of the theory book deal with the first bullet point, what a database is, so our comparative study of SQL starts with Section 1.6. (Note, by the way, that I wrote “an SQL”, not “a SQL”. Some people pronounce the name as “sequel” but the official pronunciation is “ess cue ell”.)

Historical Note

The origins of SQL can be traced to an IBM research paper published in 1974, titled *SEQUEL: A Structured English Query Language*, by Donald D. Chamberlin and Raymond F. Boyce. Raymond Boyce, who sadly died in the same year, 1974, was the person whose name is enshrined in *Boyce-Codd Normal Form* (BCNF), described in Chapter 7 of the theory book. This paper actually said nothing about database management but it assumed the existence of relation-like things called tables and defined a notation for expressing queries on tables in the declarative manner that is expected for relational databases.

A prototype implementation of SEQUEL, named SEQUEL-XRM, was produced in 1974–75. Experience with this prototype led in 1976–77 to a revised version of the language, SEQUEL/2, subsequently renamed SQL for legal reasons. Work then began on System R, another, more ambitious prototype, implementing a subset of that revised version. A significant aim of System R was to gainsay those pundits who were sceptical about the possibility of providing a relational DBMS with performance comparable to that obtainable with conventional database technology at the time, and it was widely lauded as having succeeded in that aim. System R became operational in 1977 and was installed at a number of internal sites in IBM and at a few selected customer sites. Experiences with System R resulted in further changes to SQL and soon expectations were high that IBM would eventually develop commercial SQL products—so high that one vendor organization, now named Oracle Corporation, beat IBM to it in 1979 with the first release of ORACLE. IBM’s own first SQL product, SQL/DS for the VSE environment, appeared in 1981 and implementations on other platforms followed during the 1980s. One of these was DB2 for OS/MVS in 1983, and much later DB2 became the generic name for all of IBM’s implementations. A host of other vendors joined the game over the next two decades, including, to name but a few, Microsoft (with SQL Server, naughtily pronounced “sequel server”!), Sybase, Ingres, and Digital Equipment Corp. (with RDB, now owned by Oracle). Those last two, Ingres and RDB, originally used their own languages, both inspired by Codd’s relational calculus, but later acquired an SQL interface.

The first edition of the international (ISO) standard for SQL appeared in 1987 as a ratification of one produced a year earlier by the Database Committee X3H2 of the American National Standards Institute (ANSI). Further development of the standard has taken place since then in ISO—or rather, under the auspices of a “joint technical committee”, JTC 1, of ISO and IEC, responsible for all international IT standards. The editions are referred to as SQL:yyyy, where yyyy is the year of publication.

The so-called “referential integrity” feature, supporting primary keys and foreign keys, appeared in SQL:1989. SQL:1992 contained so many additional features that its language had been referred to at the time as SQL/2. In 1996 additional features known as SQL/PSM (a programming language for stored procedures and functions) and SQL/CLI (a “call level interface”) were added. At the same time, the committee was working on further extensions to the language, resulting in the appearance of “SQL/3” as SQL:1999, whose most significant addition was the so-called “object-relational” feature supporting user-defined types. Further revisions appeared in 2003, 2007, and 2011. That last one, SQL:2011 (reference [15]), is notable for its modicum of support for “temporal database” management.

Many of the additional features appearing in post-1989 editions are specified as optional, meaning that they aren’t required for an implementation to qualify as standard-conforming.

Historical notes appearing later in this book, concerning specific features of SQL, refer mostly to the pre-2000 editions of the ISO SQL standard: 1987, 1989, 1992, 1996, 1999, though in connection with the notation for queries I can sometimes take you right back to the 1974 SEQUEL paper.

Now, Sections 1.2 to 1.4 in the theory book describe some concepts that apply to databases in general and therefore equally well to relational databases and SQL databases. For that reason counterparts of those sections are omitted here and the next section is numbered 1.5 to maintain the parallels. Also, for a similar reason, the first two Figures in this chapter are number 1.2 and 1.4.

1.5 “Collection of Variables”

Figure 1.2 in the theory book puts a variable name on a table that is later confirmed as depicting a relation. The same figure serves here to put the same variable name on a picture in tabular form representing an SQL table. To speak of a table being depicted in tabular form isn't as silly as it may seem. There are other ways of visually representing the abstract concept of an SQL table, just as there are other ways of representing relations visually. In fact, it was the suitability of such pictures that led to the terms table, row, and column being advanced as alternatives for relation, tuple, and attribute back in the 1970s.

ENROLMENT

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 1.2: An SQL table variable, showing its current value.



HELT GRATIS!

DU FÅR BOKA HOS DNB

S for Skikk & Bank

En bok om ting som er greit å vite når du har flyttet hjemmefra.

dnb.no

DNB

Bank fra A til Å



1.6 What Is an SQL Database?

So, an SQL database is one whose symbols are organized into a collection of *tables*.

Now, Figure 1.2 shows an SQL table as the current value of an SQL variable, `ENROLMENT`, but in that connection there are two important points to be made right away. First, the term *table variable* isn't used in SQL for its counterpart of relation variables. Instead, base relvars are normally referred to as *base tables*, virtual relvars as *views* (or *viewed tables*) and those terms are used in the international standard. Secondly, not every SQL table can be the value of a base table or view. As we shall soon see, table variables and the tables assigned to them are subject to several deviations from relational theory; tables that cannot be assigned to a variable are subject to even more deviations. These deviations, and their consequences, give rise to much of the subject matter in this book.

The theory book's study of relational theory includes

- the “anatomy” of a relation
- **relational algebra**: a set of mathematical operators that operate on relations and yield relations as results
- **relation variables**: their creation and destruction, and operators for updating them
- **relational comparison operators**, allowing **consistency** rules to be expressed as **constraints** (commonly called **integrity constraints**) on the variables constituting the database

and the book shows “how these, and other constructs, can form the basis of a **database language** (specifically, a *relational* database language)”. To parallel the above structure, our study of SQL includes

- the “anatomy” of a table—showing similar components to those of relations but using different terminology
- SQL's operators that operate on tables and yield tables as results
- base tables: their creation and destruction, and operators for updating them
- various special constructs allowing consistency rules to be expressed as constraints on the base tables constituting the database

Historical Note

As already mentioned, during the 1970s the term “table” became widespread as a more intuitive replacement for the mathematical term “relation”. For example, it was used in ISBL, whose developers are my dedicatees in the theory book, and also in Business System 12, the relational DBMS produced in 1982 for users of IBM’s time-sharing services. Nowadays, however, as those earlier relational DBMSs have left the scene, the term tends to refer specifically to SQL tables and as these differ significantly from relations in several important respects it is perhaps better to revert to E.F. Codd’s terminology for relations, as I do in the theory book.

1.7 “Table” Not Equal to “Relation”

The corresponding section in the theory book is titled “**Relation**” Not Equal to “**Table**”, where the word table was not intended to refer to the SQL construct in particular. Here it does refer to that construct.

The table shown in Figure 1.4 is a copy of its counterpart in the theory book, where it illustrates the point that two tables that differ only in the order of rows or the order of columns represent the same relation.

Name	StudentId	CourseId
Devinder	S4	C1
Cindy	S3	C3
Anne	S1	C1
Boris	S2	C1
Anne	S1	C2

Figure 1.4: Not the same SQL table as Figure 1.2.

In standard SQL, the order of rows still carries no significance but the order of columns does carry significance, so Figures 1.2 and 1.4 do not in fact depict exactly the same SQL table. In the course of our study we will discover how the order of columns affects the SQL operators for operating on tables and those for operating on base tables. (Some SQL implementations support a somewhat unsound operator that delivers the table consisting of the first n rows of a given table. The user can specify an ordering for the rows but is not required to do so. The unsoundness arises even in the case where the user does specify an ordering, when there is a tie for any of the first n places. The order of the rows involved in the tie is then indeterminate and results are unpredictable.)

Also in the theory book, Figure 1.5 shows a table, copied here, that does not depict any relation. However, it does depict a valid SQL table.

A	B	A
1	2	3
4		5
6	7	8
9	9	?
1	2	3

Figure 1.5: A possible SQL table.

The single exercise for Chapter 1 of the theory book invited you to list the features seen in this table that mean it cannot possibly represent a relation. One of these—the appearance of two identical column headings—means that this SQL table cannot be the value of a base table, as we shall soon see. We shall also discover how it can happen that the same row appears more than once in an SQL table, like the first and last rows in Figure 1.5, and how something other than a value, or even nothing at all, might appear in a place where a value is expected, as depicted in the second and fourth rows.

1.8 Anatomy of a Table

Figure 1.6 shows the terminology used in SQL to refer to parts of the structure of a table.

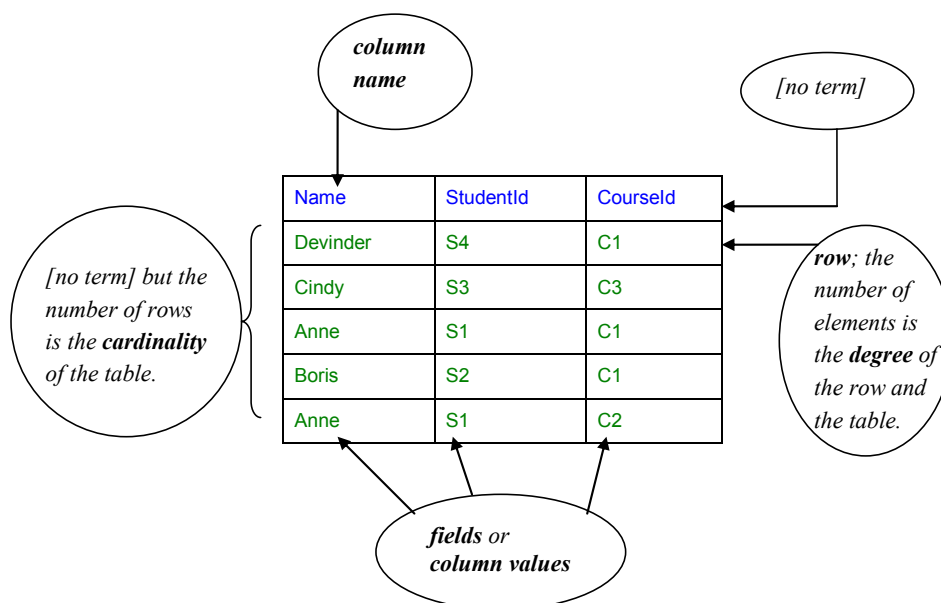


Figure 1.6: Anatomy of an SQL table.

As you can see, SQL has no official terms for its counterparts of the heading and body of a relation, but for convenience I do use those terms occasionally in the present book as applying to tables. SQL's counterpart of relational *attribute* is *column*. The figure shows the term *field* being used for SQL's counterpart of *attribute value* but please note that the international standard uses this term not only for a component of a row, as shown, but also for the corresponding component of the relevant *row type*.

1.9 What Is a DBMS?

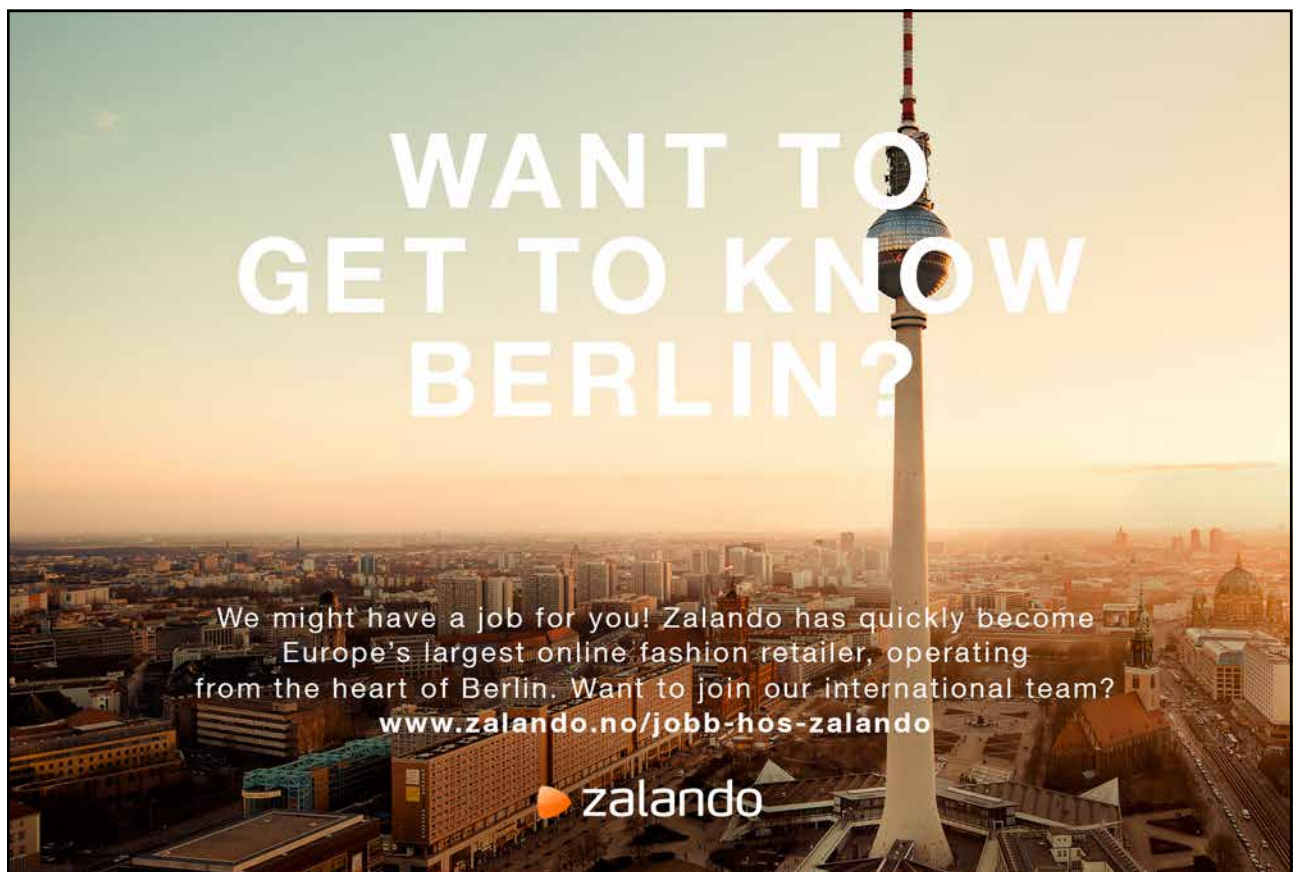
The answer to this question is of course given in Section 1.9 of the theory book.

This book is concerned with SQL DBMSs and SQL databases in particular. Soon we will be looking at some of the features we might expect to find in an SQL DBMS—“might” because not all the standard features we describe are found in all SQL implementations.

1.10 SQL Is a Database Language


Section 1.10 in the theory book begins

[The] commands given to a DBMS by an application are written in the database language of the DBMS. The term *data sublanguage* is sometimes used instead of database language. The sub- prefix refers to the fact that application programs are sometimes written in some more general-purpose programming language (the “host” language), in which the database language commands are embedded in some prescribed style. Sometimes the embedding style is such that the embedded statements are unrecognized by the host language compiler or interpreter, and some special *preprocessor* is used to replace the embedded statements by, for example, CALL statements in the host language.



WANT TO
GET TO KNOW
BERLIN?

We might have a job for you! Zalando has quickly become Europe's largest online fashion retailer, operating from the heart of Berlin. Want to join our international team?
www.zalando.no/jobb-hos-zalando

 zalando



Those remarks apply to SQL, as does the remainder of that section. Moreover, Part 10 of the international standard for SQL defines classes for use in object-oriented languages, in particular, Java, thus allowing method invocations to be used in place of old-fashioned CALL statements.

1.11 What Does an SQL DBMS Do?

In response to requests from application programs, an SQL DBMS can, for example,

- create and destroy base tables
- take note of integrity rules expressed as *constraints*
- take note of *authorisations* (who is allowed to do what, to what)
- update variables (honouring constraints and authorisations)
- provide results of *queries*

—in other words, all of the things mentioned in the corresponding list of bullet points in Section 1.11 of the theory book.

In the remaining sections of this chapter you will see examples of how an SQL DBMS does these things (via commands—usually called statements—written in SQL, of course).

1.12 Creating and Destroying Base Tables

Example 1.1 shows an SQL command to create the base table counterpart of the ENROLMENT variable shown in Figure 1.2.

Example 1.1: Creating a base table.

```
CREATE TABLE ENROLMENT
( StudentId  SID ,
  Name       VARCHAR(30) ,
  CourseId   CID ,
  PRIMARY KEY ( StudentId, CourseId )
) ;
```

(Example 1.1 and the other examples in this chapter all end in semicolons. Actually, a semicolon is required in SQL only in scripts consisting of more than one statement. A script consisting of a single statement must not end in a semicolon.)

Explanation 1.1:

CREATE TABLE is SQL's counterpart of the key words **VAR**, **BASE**, and **RELATION** as used in **Tutorial D**.

ENROLMENT is the *table name* for the base table.

The text from the opening parenthesis to the end of the fourth line specifies *the declared type* of the table, meaning that every table ever assigned to **ENROLMENT** must be a table of that type.

The declared type of **ENROLMENT** is a *table type*, indicated by the key word **TABLE** and a comma-separated list (commalist, for short) of *column definitions*. A column definition consists of a column name followed by a type specification. Thus, each column of the table also has a declared type. Two or more columns can have the same type but *not* the same name. The type names **SID** and **CID** (for student ids and course ids) refer either to user-defined *types* or to user-defined *domains*. User-defined types and domains have to be defined by some user of the DBMS before they can be referred to. The type name **VARCHAR(30)** (character strings of length 30 or less), by contrast, is a *predefined* type: it is provided by the DBMS itself, is available to all users, and cannot be destroyed.

Chapter 2, “Values, Types, Variables, Operators”, deals with types in more detail, and shows you how to define types and domains.

PRIMARY KEY indicates that the table is subject to a key constraint, in this case declaring that no two rows in the table assigned to **ENROLMENT** can ever have the same combination of **StudentId** and **CourseId** fields (i.e., we cannot enrol the same student on the same course more than once, so to speak). We will learn more about SQL constraints in general and SQL's key constraints in particular in Chapter 6.

Destruction of **ENROLMENT** is the simple matter shown in Example 1.2,

Example 1.2: Destroying a base table.

```
DROP TABLE ENROLMENT ;
```

After execution of this command the base table no longer exists and any attempt to reference it is in error.

Historical Note

The first SQL products, and indeed, the first edition (1987) of the international standard, had no support for key constraints. The `PRIMARY KEY` construct, along with the other features of the so-called “referential integrity” enhancements, appeared in SQL:1989. Because key constraints could not previously be expressed at all, by the time the feature was added there were plenty of existing base table definitions that lacked them and in such base tables it was even permitted for the same row to appear more than once (with what significance? you may well ask—good question!). As a consequence, the declaration of a key constraint had to be made optional in the interests of backwards compatibility. This is just one of many dubious features of SQL that arise from misjudgments made in the original language that have been impossible to correct for that reason.

1.13 Taking Note of Integrity Rules

For example, suppose the university has a rule to the effect that there can never be more than 20,000 enrolments altogether. Example 1.3 shows how to declare the corresponding constraint in standard SQL, but please note that not many SQL implementations actually support this optional feature at the time of writing (2012).

Example 1.3: Declaring an integrity constraint.

```
CREATE ASSERTION MAX_ENROLMENTS
    CHECK ( ( SELECT COUNT(*) FROM ENROLMENT ) <= 20000 ) ;
```

Explanation 1.3:

- **ASSERTION** is the key word indicating that a constraint is being declared.
- **MAX_ENROLMENTS** is the name of the constraint.
- **(SELECT COUNT(*) FROM ENROLMENT)** is an SQL expression yielding the cardinality of (number of rows in) the current value of ENROLMENT. Note that the enclosing parentheses are actually required: without them, the expression would denote a certain table rather than a numerical value. See Chapter 5, Section 5.3, **Aggregate Operators**, for more details on this construct.
- **(SELECT COUNT(*) FROM ENROLMENT) <= 20000** is a truth-valued expression—a *condition*, yielding *true* if the cardinality is less than or equal to 20000, otherwise yielding *false*. As we shall see later, not all conditions are such that they always yield either *true* or *false*—most of them aren’t—but this is a special case that does happen to adhere to the usual rule of logic.
- Enclosing the condition in parentheses preceded by the key word **CHECK** is necessitated because there are other things that can be optionally specified in connection with a constraint.

The declaration tells the DBMS that the database is *inconsistent* if the condition of `MAX_ENROLMENTS` is ever *false*, and that the DBMS is therefore to reject any attempt to update the database that, if accepted, would bring about that situation.

Example 1.4 shows how to retract a constraint that ceases to be applicable.

Example 1.4: Retracting an integrity constraint.

```
DROP ASSERTION MAX_ENROLMENTS ;
```

Historical Note

`CREATE ASSERTION` first appeared in SQL:1992 but it remains (in SQL:2011) an optional conformance feature and its uptake by SQL vendors has been negligible. With it, SQL's support for database integrity would be almost complete in the sense described in the theory book ("almost", because it still has no counterparts of those relations of degree zero, `TABLE_DEE` and `TABLE_DUM`); without it that cannot be so, as explained in Chapter 6.

1.14 Taking Note of Authorisations

As relational theory is silent on the issue of authorisation, it offers nothing with which SQL's vast edifice in support of what it calls *privileges* can be compared. Example 1.5 is a very simple case showing how the corresponding example in the theory book could be done in SQL.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"

Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



Example 1.5: Granting privileges

```
GRANT SELECT, INSERT, DELETE, UPDATE ON ENROLMENT TO User9 ;
```

Explanation 1.5:

- **GRANT** is the key word indicating that privileges are being granted.
- **SELECT, INSERT, DELETE, UPDATE** identify operators that can be applied to a base table. The first, **SELECT**, is used here to indicate that permission to access the current value is being granted. The others are all update operators with very much the same effects as their counterparts in **Tutorial D**. If **DELETE** were omitted, for example, then *User9* would not be allowed to use **DELETE** commands on **ENROLMENT**.
- **ON ENROLMENT** identifies the base table on which those privileges are being granted.
- **TO** is the key word required to precede the commalist of user names, officially termed *authorization identifiers*, denoting the users on whom the specified privileges are to be conferred.

So, our example actually grants four distinct privileges, all on the same base table. Example 1.6 shows how, for example, just two of these can be withdrawn (*revoked* is the official term):

Example 1.6: Revoking privileges

```
REVOKE DELETE, UPDATE ON ENROLMENT FROM User9 ;
```

Note that SQL does not use names for privileges, as suggested in the theory book's Example 1.5. If it did, then the syntax for *revoking* privileges might be simpler, but then we would need to provide a name for each of the four distinct privileges being granted in Example 1.5 and use two of those names in Example 1.6. The syntax for assigning names would surely be rather cumbersome—and consider that in real life the list of user names following **TO** might be very large.

These two simple examples hardly scratch the surface of the “vast edifice” I referred to. Furthermore, it is doubtful whether any university course in SQL would cover the whole of that edifice as defined in the international standard. Mercifully, we shall have nothing more to say about it in this book.

1.15 Updating Variables

For *assignment*, SQL uses the key word **SET**, as in **SET X = X + 1** (read as “set X equal to X+1”) rather than **X := X + 1** as found in many computer languages.

When the target is a base table, direct assignment is not available in SQL. Only differential update operators such as the usual **INSERT, DELETE, and UPDATE** are available. Example 1.8 illustrates SQL's **DELETE**.
Download free eBooks at bookboon.com

Example 1.8: Updating by deletion

```
DELETE FROM ENROLMENT WHERE StudentId = SID ( 'S4' ) ;
```

As you can see, this differs from Example 1.8 in the theory book only by the addition of the noise word `FROM`. The expression `SID ('S4')` assumes the existence of a user-defined operator whose invocation here yields the value of user-defined type `SID` that represents the student id `S4`.

Next, in Example 1.9, we look at `UPDATE`.

Example 1.9: Updating by replacement

```
UPDATE ENROLMENT SET Name = 'Ann'
      WHERE StudentId = SID ( 'S1' ) ;
```

Note the use of `SET`, as already noted in connection with direct assignment to variables other than base tables. Otherwise the syntax is very similar to that used in **Tutorial D**, as are the semantics.

Finally, Example 1.10 illustrates the use of `INSERT`.

Example 1.10: Updating by insertion

```
INSERT INTO ENROLMENT
      VALUES ( SID ( 'S4' ) ,
              'Devinder' ,
              CID ( 'C1' ) ) ;
```

Here we see our first consequence of the difference between relations and SQL tables. The key word `VALUES` is SQL's counterpart of `RELATION` as used in relation selector invocations in **Tutorial D** (see Example 1.10 in the theory book). Most importantly, note that the expression denoting the operand of `VALUES` specifies fields (column values) without giving their column names, raising the question, how does SQL know which value goes into which column? The answer, as I'm sure you've guessed, is that the first value, `SID ('S4')`, goes in the first column of `ENROLMENT`, the second in the second, and so on. That in turn raises the question, how do we know which is the first column of `ENROLMENT`, which the second, and so on, considering that in relational theory there is no ordering to the attributes of a relation? The answer, as I'm sure you've guessed again, lies in the order in which the column definitions are listed in the invocation of `CREATE TABLE` that brought `ENROLMENT` into existence (Example 1.1). However, SQL does make provision in case you remember the column names but forget the order: you could write the names in parentheses, `(StudentId, Name, CourseId)`, the order corresponding to that of the `VALUES` entries, after the table name `ENROLMENT`.

A few words are needed to say what happens when rows to be deleted or updated do not exist and rows to be inserted already exist.

Example 1.8 has no effect on the database in the case where the current value of `ENROLMENT` has no rows for student `S4`, but the DBMS might give a warning message.

Example 1.9 has no effect on the database in the case where the current value of `ENROLMENT` has no rows for student `S1`, but again the DBMS might give a warning message.

Regarding Example 1.10, in the case where the current value of `ENROLMENT` already contains the row `(SID ('S4') , 'Devinder' , CID ('C1'))`, the effect is to add another copy of that row anyway, unless to do so would cause some explicitly declared constraint (perhaps a key constraint) to be violated. By the way, the outer parentheses enclosing the three expressions denoting that row are required only when the row contains more than one field. Somewhat counterintuitively, the expression `VALUES 1, 2` denotes a table having two rows and just one column, as does `VALUES (1), (2)`, whereas `VALUES (1, 2)` denotes a table with one row and two columns.

Historical Notes

`INSERT`, `DELETE`, and `UPDATE`, including the use of `SET name = expression` for specifying column assignments, all appeared in the first SQL products. Support for local variables arrived in SQL:1996 with SQL's computationally complete programming language, SQL/PSM. It was natural, then, to use the same syntax for assignment to variables. Had the need for a programming language been realized at the outset, then perhaps a more usual syntax for assignment might have been adopted and copied in column assignments in `UPDATE` commands. That said, it can be noted that Basic uses similar syntax to SQL, replacing `SET` by `LET`.

The curious syntax for `VALUES` expressions is explained by the fact that they were originally restricted to just one row and could appear only as the source operand for `INSERT`. For imagined convenience, the parentheses surrounding the list of field expressions could be omitted when the row consisted of a single field. In SQL:1992 the syntax was extended to allow more than one row to be specified, using commas to separate the individual, possibly parenthesized, row expressions. However, that extension was specified as an optional conformance feature and it remains optional in SQL:2011. Use of a `VALUES` expression other than as an operand of `INSERT` is also an optional conformance feature.

1.16 Providing Results of Queries

Expressing queries in SQL is the (big) subject of Chapters 4 and 5. Here I present just a simple example to give you the flavour of things to come in those chapters. Example 1.11, as in the theory book, is a query expressing the question, who is enrolled on course C1?

Example 1.11: A query in SQL

```
SELECT DISTINCT StudentId, Name
FROM   ENROLMENT
WHERE  CourseId = CID ( 'C1' )
```

Note carefully that Example 1.11 is not a command (hence the absence of a semicolon). It is just an expression, denoting a value—in this case, a table. In SQL the result of a query is always another table. Figure 1.7 shows the result of Example 1.11 in the usual tabular form.

StudentId	Name
S1	Anne
S2	Boris
S4	Devinder

Figure 1.7: Result of query in Example 1.11.

Explanation 1.11:

- **SELECT DISTINCT StudentId, Name** specifies that the result of the **WHERE** invocation is to be projected over **StudentId** and **Name**.
- **FROM ENROLMENT** specifies the table operand for the invocation of **WHERE**.
- **WHERE CourseId = CID ('C1')** specifies that just the rows for course C1 are required.

Historical Note

The syntax illustrated here is very similar to that defined for SEQUEL back in 1974. A table expression is required to begin with a **SELECT** invocation. That invocation or, to give its official term, *clause*, is followed by a sequence of clauses, each denoting a table that is the single table operand of the clause that follows it. The table denoted by the last clause is the table operand of the **SELECT** clause. The second clause must always appear and must always be a **FROM** clause. If a **WHERE** clause appears at all, it must be the third one. We shall meet other clauses, and the rules governing their possible appearance, later. These rules define the *structure* that inspired the name of the language when SEQUEL morphed into SQL, though in the title of the SEQUEL paper it referred to the ability to nest such expressions inside other such expressions. Officially the name is just SQL, not standing for anything—neither “structured query language” nor, as some would have it, “standard query language”).

Download free eBooks at bookboon.com


2 Values, Types, Variables, Operators

2.1 Introduction

This chapter in the theory book looks at the four fundamental concepts on which most computer languages are based and establishes the related terminology that is used in the book. Here we examine, at a general level, SQL's treatment of those four concepts, but unfortunately SQL doesn't stick with just those four and in fact has two more. One of these is so pervasive in its effects on the other four that we had better start to examine it right away...

SQL's Fifth Concept, NULL

Unfortunately, SQL embraces a fifth concept, called `NULL`, an apparently simple little thing but one that has pervasive effects on our usual understanding of the other four. `NULL` denotes a kind of nothingness. It is somewhat akin to a value in that it can be the result of evaluating an expression, but it has no type, cannot appear everywhere a value can appear, and lacks certain other properties that values have, as we shall see.



WHILE YOU WERE SLEEPING...

www.fuqua.duke.edu/whileyouweresleeping

DUKE
THE FUQUA
SCHOOL
OF BUSINESS



Although SQL is not the only computer language to include a special construct representing non-existence, its own variety, `NULL` is accompanied by a departure from classical logic that is not found in any other well-known languages and gives rise to some deviations from relational database theory that students and users have to be well aware of. The departure in question is the introduction of a third truth value, denoted by the key word `UNKNOWN`. For example, a comparison of the form $x = y$ in SQL evaluates to `UNKNOWN` if either x or y evaluates to `NULL`, even when they both do! (The comparison operator `IS NOT DISTINCT FROM` is available in place of `=`, such that $x \text{ IS NOT DISTINCT FROM } y$ evaluates to `FALSE` if just one of x and y is `NULL`, `TRUE` if they both are, and is otherwise equivalent to $x = y$.)

I shall reveal as we go along the effects of `NULL` and `UNKNOWN` that cause SQL to deviate from normally expected behaviour.

Historical Note

SQL's `NULL` and `UNKNOWN` are generally considered to have their origins in a 1979 paper by E.F. Codd (reference [6]), who was an employee of IBM at that time and thus readily available for consultation by the early SQL DBMS developers in that company. It is perhaps rather unfortunate, then, that these constructs, which have been subject to so much criticism and controversy, thus received a cachet of approval from such an eminent authority. In his famous 1970 paper (reference [3]) Codd was quite clear in requiring every attribute of a tuple contained in a relation to be assigned a value in the “domain” (i.e., declared type) of that attribute. It seems likely (to me, at least) that in 1979 Codd was reacting to critics of his Relational Model of Data who complained that it appeared to offer no good, practical method of dealing with the so-called “problem of missing information”, though an earlier paper by him (reference [4]) does contain a couple of oblique mentions of something called “null”. That criticism might still be to some extent valid. For example, several genuinely relational approaches to the problem that have been advanced are described in reference [11], yet none of these is likely to appeal to those who are beguiled by the superficial attraction of just “leaving it blank”, so to speak.

It is interesting that Codd later rejected his 1979 proposal as being inadequate (rather than as being flawed, as some would have it) and replaced it in 1990 (reference [5]) by a proposal involving two different kinds of null and a fourth truth value. The point was that his original proposal was for a special marker to appear, in place of an attribute value, to denote “a value exists but which particular value really belongs here is not known”. SQL's `NULL` is indeed used for that purpose but, as we shall see, it is used for other purposes too. One of those other purposes is to signify “no value belongs here”, and that is the additional purpose Codd was addressing in 1990.

As we shall see as we go along, the beguiling simplicity of “just leaving it blank” (i.e., assigning `NULL`) is more than compensated for—its detractors such as myself would argue—by the complications and difficulties that arise, for example, in defining constraints and expressing queries.

Now, although I have claimed that `NULL` is a “fifth concept”, the current (2011) SQL standard’s definition of `NULL` appears to directly contradict that claim, and also to take a different view on my claim that `NULL` has no type:

Every data type includes a special value, called the *null* value, sometimes denoted by the keyword `NULL`. This value differs from other values in the following respects:

- Since the null value is in every data type, the data type of the null value implied by the keyword `NULL` cannot be inferred; hence `NULL` can be used to denote the null value only in certain contexts, rather than everywhere that a literal is permitted.
- Although the null value is neither equal to any other value nor not equal to any other value — it is *unknown* whether or not it is equal to any given value — in some contexts, multiple null values are treated together; for example, the `<group by clause>` treats all null values together.

Note the term *the null value*, and that although `NULL` is called that, it “differs from all other values”. In particular it compares neither equal to itself nor unequal to everything bar itself, except in certain contexts as vaguely noted in the second bullet. `IS NOT DISTINCT FROM` is one of these exceptions. It, and its converse `IS DISTINCT FROM`, were added to SQL in 1999 and they remain an optional conformance feature. Note also the notion that there is only one `NULL`, causing us to wonder how it can be that the `NULL` that is a member of type `INTEGER` is the very same thing as the `NULL` that is a member of type `DATE`, for example. I stick by my “fifth concept” claim.

I have also mentioned `UNKNOWN` as the key word denoting SQL’s third truth value. In the light of the text I have just cited, to the effect that `NULL` is a value of every type in SQL, the question should arise in your mind: what is the difference between `UNKNOWN` and the `NULL` that is a member of type `BOOLEAN`? Here is the official answer (again, as of 2011):

The data type boolean comprises the distinct truth values *True* and *False*.... [T]he boolean data type also supports the truth value *Unknown* as the null value. This specification does not make a distinction between the null value of the boolean data type and the truth value *Unknown*...they may be used interchangeably to mean exactly the same thing.

Notice in passing “the null value of the boolean data type”, strongly suggesting that this is not, after all, the very same thing as the null value of the integer data type. Clearly, `NULL` is a very strange and difficult thing to speak or write about, especially for the painstaking members of the international committee responsible for drafting the SQL standard (and I can attest to that difficulty, having been one of them for 15 years!).

2.2 Anatomy of A Command

Figure 2.1, showing a simple SQL command, is almost identical to its counterpart in the theory book. The only difference arises from the fact that SQL uses a different notation for assignment to a local variable. (The SQL international standard doesn't actually use the terms *update operator* and *read-only operator*, but SQL does embrace those distinct concepts and for convenience I shall continue to use those terms in this book.)

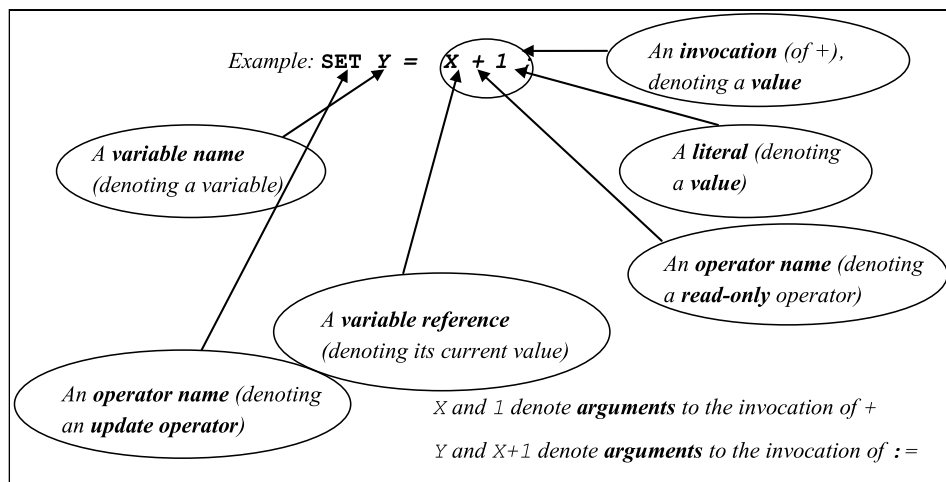


Figure 2.1: Some terminology.



Vi vokser i Norge og har virksomhet helt frem til 2050

Er du interessert i sommerjobb
eller fast stilling?



Se informasjon om sommerjobber på
www.bp.no





The remaining text of Section 2.2 in the theory book holds good here, subject to the two small syntactic differences, the initial key word `SET` as the operator name for assignment and the use of `=` in place of `:=`.

Effects of `NULL`

The numeric variable `X`, perhaps of type `INTEGER`, might be assigned `NULL`. In that case the result of evaluating `X + 1` is `NULL`, and so `SET Y = X + 1` assigns `NULL` to `Y`. A curious consequence of this is that immediately following this assignment the comparison `Y = X + 1` does not evaluate to `TRUE`. For that matter, neither does the comparison `Y = Y`. In fact they both evaluate to `UNKNOWN`.

2.3 Important Distinctions

Again the theory book text for this section holds good here too. The list of important distinctions is repeated here. You might wish to check your own understanding of them before re-reading the text in the theory book.

- Syntax versus semantics
- Value versus variable
- Variable versus variable reference
- Update operator versus read-only operator
- Operator versus invocation
- Parameter versus argument
- Parameter subject to update versus parameter not subject to update

2.4 A Closer Look at a Read-Only Operator (+)

In the theory book I equate the term *read-only operator* to the mathematical term *function*. Here I just need to add that the SQL standard reserves the term *function* for read-only operators that are invoked using prefix notation: an operator name followed by a commalist of argument expressions enclosed in parentheses, as in for example `SQRT (2)`, denoting the positive square root of 2, or `SUBSTRING (X FROM 1 FOR 2)` denoting the first two characters of the character string denoted by `X`. The term *operator* is used for those invoked using infix notation, like `+` for example.

2.5 Read-only Operators in SQL

Like **Tutorial D** and many other computer languages, SQL distinguishes between *system-defined* (or *built-in*) operators and *user-defined* operators.

I do not give a list of SQL's system-defined operators. For one thing it would be too overwhelming for present purposes. For another, it would inevitably be incomplete sooner or later, as the SQL standard is revised every few years and implementers are naturally inclined to add new ones on demand. We will meet some of them when we need them for illustrative purposes.

Section 2.5 of the theory book gives Example 2.1, defining in **Tutorial D** an operator named `HIGHER_OF` to give the value of whichever is the higher of two given integers. Here I simply translate that definition into SQL.

Example 2.1: A User-Defined Operator in SQL

```
CREATE FUNCTION HIGHER_OF ( A INTEGER, B INTEGER )
    RETURNS INTEGER
IF A > B THEN RETURN A ;
    ELSE RETURN B ;
END IF ;
```

Points to note:

- the key word `FUNCTION` in place of **Tutorial D's** `OPERATOR`;
- some minor differences concerning semicolons;
- the lack of a counterpart to **Tutorial D's** `END OPERATOR` (which some observers have criticised as being redundant);
- above all, the initial key word, `CREATE-SQL` uses this for all sorts of database objects that users can create. For example, `CREATE TABLE` is SQL's counterpart of **Tutorial D's** `VAR ... BASE RELATION` for creating a database relvar, and `CREATE PROCEDURE` is for creating a user-defined update operator.

Example 2.1 doesn't illustrate all of the features in SQL connected with user-defined operator definitions. For example, the code that implements the operator can be written separately, in a language other than SQL. Also, the key word `FUNCTION` can be replaced by some special syntax to denote a special kind of function referred to as a *method* (as used in several object-oriented programming languages), but that and certain other optional ingredients are beyond the scope of this book.

Effects of NULL

As a general rule—but not a universal one—if NULL is an argument to an invocation of a system-defined read-only operator, then NULL is the result of that invocation. As you can see, the code for `HIGHER_OF` includes `A > B`, an invocation of the system-defined read-only operator “>”, which does follow the general rule. Hence, if NULL is substituted for either or both of the parameters A and B, then NULL—which in this case we can also call UNKNOWN because “>” is a Boolean operator—is the result of the invocation. You are perhaps now wondering how SQL handles the IF statement when the specified condition yields UNKNOWN: is the THEN clause evaluated, or is it the ELSE clause?

As you know, other programming languages are normally based on classical logic. In keeping with the existence of just two truth values, TRUE and FALSE, the syntax for IF statements (and IF expressions) in such languages has just the two forks, THEN for when the condition is TRUE, ELSE for when it is not (i.e., is FALSE). You might therefore reasonably expect a language that embraces *n* truth values to support a variety of IF that has *n* forks—under a language design principle that Fred Brooks in reference [1] referred to as *conceptual integrity*, which means adhering rigorously to the language’s adopted concepts. Instead, SQL retains just the two forks, keeping the normal treatment of THEN as being the one for when the condition is TRUE and arbitrarily lumping UNKNOWN in with FALSE for the ELSE fork.

You should now be able to see that the general rule (“NULL in, NULL out”) for system-defined operators cannot be said to apply to user-defined ones. If `A > B` evaluates to UNKNOWN, then the result of the `HIGHER_OF` invocation is the argument substituted for B, which might or might not be NULL.

If we wanted to make `HIGHER_OF` adhere to “NULL in, NULL out”—let’s call it the NiNo rule—we would have to write something like what is shown in Example 2.1a.

Example 2.1a: NiNo version of HIGHER_OF

```
CREATE FUNCTION HIGHER_OF ( A INTEGER, B INTEGER )
    RETURNS INTEGER
CASE
    WHEN A IS NULL OR B IS NULL
        THEN RETURN CAST ( NULL AS INTEGER ) ;
    WHEN A > B
        THEN RETURN A ;
    ELSE RETURN B ;
END CASE ;
```

Explanation 2.1a:

- **IS NULL** is a monadic Boolean operator that evaluates to **TRUE** when its argument is “the null value”, otherwise **FALSE**. Note, therefore, that it is our first exception to the NiNo rule, which would require it to evaluate to **NULL** (**UNKNOWN**) when its argument is “the null value”.
- **CAST (NULL AS INTEGER)** denotes “the null value” of type **INTEGER**. As in **Tutorial D**, the operand of **RETURN** must be an expression that has a declared type and **NULL**, on its own, does not have a declared type. The **CAST** operator takes an expression and a type name, separated by the “noise” word **AS**, and normally expresses a “type conversion”—a function that maps elements of one type to those of another that are considered to be in some defined sense equivalent. In the special case of **NULL** it is used to confer a type, so to speak, on something that doesn’t otherwise have one.
- **AS**, appearing where you might have expected just a comma, is explained by a matter of policy in SQL whereby invocations of user-defined functions, which always use commas between arguments, can be syntactically distinguished from invocations of system-defined functions.
- **OR** is SQL’s counterpart of the usual logical operator of that name. In **A IS NULL OR B IS NULL** its operands are clearly restricted to just **TRUE** and **FALSE** and that expression therefore results in **TRUE** if either of those operands does, otherwise **FALSE**. I deal with the treatment of **UNKNOWN** in SQL’s counterparts of the logical operators in Chapter 3, “Predicates and Propositions”.

Indeterminacy in SQL

Some SQL expressions are actually not function invocations at all in the mathematical sense, being indeterminate—invocations operating on identical input do not always yield the same value. The indeterminate expressions are among those that the standard defines as *possibly non-deterministic*, but—*caveat lector*—not all expressions defined as possibly non-deterministic are actually indeterminate from a mathematical viewpoint. For example, the key word **USER** denotes the *userid* (officially, the *authorization identifier*) of the session in which an expression containing that key word is evaluated. Such an expression is defined as possibly non-deterministic by virtue of the appearance of **USER**, even though invocations in different sessions, in which **USER** stands for different *userids*, are clearly different invocations. For a more general example, a user-defined function can be explicitly declared as either **DETERMINISTIC** or **NOT DETERMINISTIC**. In the latter case the “function” is flagged such that all invocations of it are treated as possibly non-deterministic.

Regardless of the appropriateness of the term non-deterministic, there is a good reason for categorizing references to the current user (or, for another example, the current time) along with genuine cases of indeterminacy. It concerns constraint declarations. We clearly want to outlaw a constraint condition whose result, when evaluated, depends on the properties of the session in which, or on the time at which, the evaluation takes place. Of course we must also outlaw genuinely indeterminate conditions—a database might satisfy such a condition but later fail to satisfy it even though the database has not been updated in the meantime! The question then arises as to how it is possible for indeterminacy to arise: surely a computer program always gives the same result when invoked with the same input?



One root cause of indeterminacy in SQL lies in its implementation of comparison for equality. For certain system-defined types it is possible for distinct values to compare equal (note the contradiction). One such type is `CHARACTER`, described later in Section 2.6. Like COBOL, SQL ignores trailing “pad characters” when comparing character strings. The pad character is normally the space obtained by depressing the space bar on a keyboard. Thus, for example, the comparison `'SQL' = 'SQL '` evaluates to `TRUE`, even though `CHAR_LENGTH('SQL') = CHAR_LENGTH('SQL ')`, comparing the lengths of those two strings, 3 and 6, evaluates to `FALSE`. Now consider the relational projection of `ENROLMENT` over just its `Name` attribute: `ENROLMENT{Name}` in **Tutorial D**. As we shall see in Chapter 4, SQL has a counterpart of projection, but suppose the two rows for student S1 in the `ENROLMENT` table had `'Anne'` and `'Anne '` for S1’s name. If both of those values were to appear in the result, that would be inconsistent with the fact that they compare equal in SQL. If just one of them appears, then which one? The SQL standard declares such an expression to be possibly non-deterministic and permits a conforming implementation to give any value that compares equal to `'Anne'`—possibly one that doesn’t even appear in the table—and does not require it to give the same value every time the expression is evaluated. As a consequence, there are several SQL operators whose use on character strings is not permitted to appear in constraint declarations.

The SQL standard lists a multitude of conditions that cause an expression to be defined as possibly non-deterministic. Perhaps the most alarming is the assumption that equals comparison of values of user-defined types is assumed to suffer from the same problem as I have described for character strings: it is assumed that distinct values can compare equal, even if the type definition is such that this cannot possibly be the case.

Historical Note

Support for user-defined operators, along with a programming language for their code, didn’t arrive in the SQL standard until 1996, 17 years after the appearance of the first commercial SQL product. Before that time some products had provided such support as proprietary (i.e., nonstandard) extensions. Even though those products may have later adopted the standard syntax, they will not have abandoned their nonstandard syntax and so these products may suffer from undesirable complications and redundancy as a result.

It is interesting to note that the designers of Business System 12 (reference [8]) recognised the need for user-defined operators, and a programming language to write them in, in 1978, before the first SQL product hit the scene. It’s perhaps a pity that their colleagues in the System R team didn’t take a closer look at Business System 12, considering that there was some contact between those two teams in the 1970s. In fact, the same requirement had been recognized even earlier by the designers of PRTV (reference [17]) at IBM UK’s Scientific Centre in Peterlee, who were consulted by the Business System 12 team and also in contact with the System R team.

PRTV stands for “Peterlee Relational Test Vehicle”, a notably unglamorous and unassuming title. In fact it was based on an earlier prototype—possibly the very first implementation of Codd’s ideas—developed at the same centre back in 1971 with the much more impressive name, IS/1. The switch to the more modest name was dictated by IBM, as they did not wish to appear at all committed to the relational model at that time.

2.6 What Is a Type?

SQL’s concept does not differ significantly from that defined in the theory book, apart from that business concerning `NULL`. However, the theory book equates *type* with the term *domain* used in much of the relational database literature. SQL is at odds with this equation because it uses *domain* for a defined subset of a given type that is not itself a type. For example, the domain `WEEKDAY` might be defined to consist of the values 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', and 'Saturday', but the declared type of a column defined on that domain is that on which the domain itself is defined, perhaps `VARCHAR(9)`. Also, whereas a domain is defined by specifying a constraint (on some underlying type), a constraint cannot be used to specify a user-defined type.

The advertisement for GaiTEYE features a background image of a person running on a path during a sunrise or sunset. The GaiTEYE logo, consisting of a yellow square with a stylized 'G' and the word 'gaiTEYE' in white, is in the top left. Below it is the tagline 'Challenge the way we run'. In the center-left, the text 'EXPERIENCE THE POWER OF FULL ENGAGEMENT...' is displayed in white, followed by a horizontal line of yellow dots. Below this, the text 'RUN FASTER. RUN LONGER.. RUN EASIER...' is shown in white. In the bottom right, a yellow button contains the text 'READ MORE & PRE-ORDER TODAY' and 'WWW.GAITEYE.COM', with a hand cursor icon pointing at it. A white line drawing of a foot with motion lines is positioned near the runner's foot.



The system-defined types of SQL at the time of writing are:

- `CHARACTER` or, synonymously, `CHAR`, for character strings. When this type is to be the declared type of something (e.g., a column), the permissible values are further constrained by a maximum length specification given in parentheses and optionally by the key word `VARYING`, indicating that values shorter than the maximum, including the empty string `' '`, are also permissible. Examples: `CHAR(5)` for strings of five characters only, `CHARACTER VARYING(100)`, which can be abbreviated to `VARCHAR(100)`, for strings of up to one hundred characters. The last two are alternative spellings for the same declared type. The type `CHARACTER LARGE OBJECT`, or `CLOB`, allows for inclusion of strings that are longer than the longest supported by the other `CHARACTER` types. Note that in the terminology of the theory book `CHARACTER` is a kind of *type generator*. The key word does not of itself denote a type, but only does so when qualified by a length specification. A similar remark applies to some of the other type names used in SQL.
- `DECIMAL`, `NUMERIC`, `REAL`, `FLOAT` and various other terms for various sets of rational numbers. When these key words are specified for the declared type of something, they are usually accompanied by *precision* and *scale* specifications. `DECIMAL` and `NUMERIC` are synonymous and are used for types with uniform scales, where the difference between any number and its immediate successor or predecessor is constant. For example, `DECIMAL(5, 2)` consists of rational numbers with a precision of 5 (i.e., having at most 5 decimal digits) and a scale of one hundredth (i.e., 2 places of decimals), such as 372.57 and -1.00. (Note that SQL always uses the period “.” as the decimal separator.)
- `INTEGER` or, synonymously, `INT`, for integers within a certain range. SQL additionally has types `SMALLINT` and `BIGINT` for certain ranges of integers. The exact sets of values denoted by these type names are not specified in the SQL standard, which states merely that “[t]he precision of `SMALLINT` shall be less than or equal to the precision of `INTEGER`, and the precision of `BIGINT` shall be greater than or equal to the precision of `INTEGER`”.
- `TIMESTAMP` for values representing points in time on a specified uniform scale. `DATE` is used for timestamps on a scale of one day, such as `DATE '2012-09-25'` (a literal denoting September 25th, 2012). `TIME` is used for values denoting time of day only, such as `TIME '16:30:00'` (a literal denoting half-past four in the afternoon).
- `INTERVAL` for values denoting, not intervals (!) but durations in time, such as 5 years, 3 days, 2 minutes, and so on.
- `BOOLEAN`, consisting of the values `TRUE` and `FALSE`, with `UNKNOWN` being an alternative name to `NULL` for “the null value” of this particular type.

- `BINARY LARGE OBJECT` for arbitrarily large bit strings.
- `XML` for XML documents and fragments.
- `ARRAY` types for arrays.
- `ROW` types and `MULTISET` types as described later in this Chapter.

At this point, having previously introduced `NULL` as SQL's "fifth concept", I need to mention SQL's sixth: *pointers*. Like `NULL`, pointers are expressly excluded from relational database theory and in fact the difficulty they give rise to was one of Codd's stated motivations for his Relational Model. A pointer is a value that somehow identifies a repository, such as a variable for example, where some other value is stored. The object identifiers (oids) used in object-oriented programming languages are pointers of a kind. SQL's so-called *REF values* are another and so are its so-called *datalinks*. A `REF` value points to a row in a base table and is otherwise rather like an oid in the operations that can be performed on it. A *datalink* is a url, allowing SQL database management to be synchronized with management of related objects that are strictly outside of the database. To complete the list of SQL's system-defined types I therefore need to add:

- `REFERENCE` types for `REF` values. A `REFERENCE` type comes into existence automatically when a certain kind of user-defined type is defined, as I describe in Section **2.10, Types and Representations**.
- `DATALINK` for *datalinks*. I say no more about *datalinks* in this book.

Historical Note

The `CHARACTER` and numeric types have been in SQL since the appearance of the first commercial products. The same is true of truth-valued expressions but `BOOLEAN` did not become a "first-class" type in the SQL standard until 1999 and it remains an optional conformance feature. (In common parlance, a type is "first-class" only if it has a name and can thus be used for any of the purposes described in Section 2.7, **What Is a Type Used For?**, in the theory book.) `TIMESTAMP` types and `INTERVAL` types arrived in standard SQL in 1992. `LARGE OBJECT` types, `DATALINK`, `ARRAY` types, user-defined types, and `REF` types first appeared in the 1999 edition. Type `XML` and `MULTISET` types were introduced in the 2003 edition.

Rows and tables obviously appeared in the original SQL products but `ROW` types as first-class types didn't appear until 1999. Table types became almost first-class with the introduction in SQL:2003 of `MULTISET` types, as described in Section 2.8, **The Type of a Table**, but SQL:2011 still does not support `TABLE` types *per se*. `ROW` types and `MULTISET` types are both optional conformance features.

2.7 What Is a Type Used For?

In SQL, as in most computer languages, a type can be used for *constraining* the values that are permitted to be used for some purpose. In particular, it can be used for constraining:

- the values that can be assigned to a variable
- the values that can be substituted for a parameter
- the values that an operator can yield when invoked
- the values that can appear in a given column of a table

In each of the above cases, the type used for the purpose in question is the *declared type* of the variable, parameter, operator, or column, respectively. As a consequence, every expression denoting a value has a declared type too, whether that expression be a literal, a reference to a variable, parameter, or column, or an invocation of an operator. Thus SQL is able to detect errors at “compile time”—by mere inspection of a given script—that would otherwise arise, and cause much more inconvenience, at run time (when the script is executed).

Unfortunately that’s not the end of the type story in SQL. SQL adds a fifth use for types, though this one is not available with all types but only for one of its two kinds of user-defined types, namely, the so-called *structured types*. A structured type is defined using something similar to **Tutorial D**’s “possrep” construct. Thus, type `POINT` might be defined in SQL in terms of a representation consisting of components `X` and `Y`, each of type `REAL`. Such a type can be used for any of the four purposes already listed but it can also be used as a basis for defining a base table (or view), consisting of columns corresponding to the components of its structure, plus one further column whose declared type is a `REFERENCE` type (`REFERENCE (POINT)` if `POINT` is the name of the structured type). The `REFERENCE` column is a relational key for the table thus defined, which is variously called a *referenceable table* and a *typed table* (the two terms are synonymous). The `REF` values can be used in the way object identifiers are used in object-oriented languages.

Effects of `NULL`

Although an SQL type does have those constraining purposes, it must be remembered that in each case `NULL` might appear in place of a true value of the type in question, except where explicit measures are taken to avoid such appearances. Appearance of `NULL` in a column of a base table can easily be avoided by specifying `NOT NULL` in the column definition, but no such declaration is available in connection with local variables, parameters, or results of operator invocations.

Historical Notes

Support for local variables didn't arrive in SQL until 1996, when support for user-defined operators first appeared. Until that time “the values that can be assigned to a variable” applied only to base tables (SQL's counterparts of base relvars), “the values that can be substituted for a parameter” applied only to parameters of system-defined operators, and “the values that an operator can yield” similarly applied only to system-defined operators.

Support for user-defined types, including the structured types mentioned in this section, was added in 1999, as part of the so-called “object-relational” support that was billed as the main theme of SQL:1999.

2.8 The Type of a Table

This section in the theory book is headed **The Type of a Relation** and it uses its running example, repeated again here as Figure 2.3 as a basis for its discussion.

StudentId	Name	Courseld
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 2.3: Enrolments again

Strømmen produseres ofte langt fra der den skal brukes.

Statnett sitt oppdrag er å gjøre strømmen tilgjengelig, uansett hvor i dette langstrakte landet du bor. Det er vi som bygger og drifter "riksveiene" i norsk strømforsyning. Gjennom vårt landsdekkende nett sørger vi for en sikker fordeling av strøm mellom nord, sør, øst og vest.

Vi binder Norge sammen

Statnett
Vårt felles kraftnett

Er du student? Les mer her
www.statnett.no/no/Jobb-og-karriere/Student



As a relation in **Tutorial D** the type name for the value shown in Figure 2.3 can be written as

```
RELATION { StudentId SID, Name NAME, CourseId CID }
```

or, equivalently (for recall that there is no ordering to the elements of a set),

```
RELATION { Name NAME, StudentId SID, CourseId CID }
```

(and so on).

Of course Figure 2.3 could also be illustrating some SQL table, but SQL has no names for table types, so no counterparts of relation type names. It does, however, have names for row types and in fact the standard uses term *row type* as a property of a table—it is of course the type of each of the rows in the table, though it is defined even when the table is empty.

A row type name is the key word ROW followed by a commalist of field name/type name pairs in similar style to the attribute name/type name pairs in **Tutorial D** except that the commalist is enclosed in parentheses rather than braces (SQL uses the term *field* for the components of a row). It would be nice, then, if we could say that the row type of our enrolments table was

```
ROW ( StudentId SID, Name NAME, CourseId CID )
```

or, equivalently,

```
ROW ( Name NAME, StudentId SID, CourseId CID )
```

but there are two important reasons why we can't. First, those two type names would in fact define different types in SQL, because the order of the elements is significant. It follows that the order of columns in an SQL table is significant, contrary to one of the important precepts you have learned in relational theory, and we will discover various complications that arise from this significance as we go along. Secondly, even if type names SID, NAME, and CID are defined, the types so named cannot be equivalent to their **Tutorial D** counterparts, for reasons given in Section 2.10, **Types and Representations**. The row type of our enrolments table is therefore more likely to be something like:

```
ROW ( Name VARCHAR(50), StudentId VARCHAR(5),  
      CourseId VARCHAR(5) )
```

SQL's row types exhibit one further deviation from relational theory: there is no such type as ROW (). It follows that every table in SQL has at least one column. TABLE_DEE and TABLE_DUM, defined in Chapter 4 of the theory book, do not exist in SQL.

MULTISET types

An SQL *multiset* is what in mathematics is also known as a *bag*—something like a set except that the same element can appear more than once. The body of an SQL table is in general a bag of rows, rather than a set of rows, because SQL does indeed permit the same row to appear more than once in the same table. Although SQL has no names for table types, it does support multisets in general and it does have names for multiset types. A multiset type name consists of a type name followed by the key word `MULTISET`. For example, `INTEGER MULTISET` is the name of the type each of whose values is either (a) a bag, consisting of zero or more appearances of each value of type `INTEGER` and zero or more appearances of the null value of type `INTEGER`, or (b) the null value of type `INTEGER MULTISET`.

It would seem at first glance, then, that we perhaps do have a type name for a table type after all. For example, our enrolments table could perhaps be of type

```
ROW ( Name VARCHAR(50), StudentId VARCHAR(5),
      CourseId VARCHAR(5) ) MULTISET
```

In fact one could declare a *local* variable to be of this type and its value could indeed consist of the rows shown in Figure 2.3. However, such a type cannot be the declared type of a base table, in spite of the fact that the elements of a base table are indeed rows of the same type. Moreover, as I have already mentioned, there is such a thing as the null value of that multiset type, whereas `NULL` can never appear in place of a table—no table expression in SQL can ever evaluate to `NULL`—nor can `NULL` appear in place of a row in a table. So the set of values of a multiset type whose element type is a row type includes bags that are not tables as well as bags that are.

2.9 Table Literals

One might expect SQL to support table literals in the manner illustrated in Example 2.2, but in fact that is not a legal SQL expression.

Example 2.2: Not a Table Literal

```
TABLE (
  ROW ( StudentId 'S1', CourseId 'C1', Name 'Anne' ),
  ROW ( StudentId 'S1', CourseId 'C2', Name 'Anne' ),
  ROW ( StudentId 'S2', CourseId 'C1', Name 'Boris' ),
  ROW ( StudentId 'S3', CourseId 'C3', Name 'Cindy' ),
  ROW ( StudentId 'S4', CourseId 'C1', Name 'Devinder' )
)
```

It is illegal because row literals in SQL do not use column names. Instead, the column values must be written in the appropriate order, reflecting the order of the columns of the table, as in

```
ROW ( 'S1', 'C1', 'Anne' )
```

Moreover, the word `VALUES` is used in place of `TABLE`, parentheses are not used around the list of row literals, and the key word `ROW` is in fact optional, so that the most common form is that shown in Example 2.3.

Example 2.3: A Table Literal (correct version)

```
VALUES
( 'S1', 'C1', 'Anne'      ),
( 'S1', 'C2', 'Anne'      ),
( 'S2', 'C1', 'Boris'      ),
( 'S3', 'C3', 'Cindy'      ),
( 'S4', 'C1', 'Devinder' )
```

Now, the question arises, what is the (table) type of the table shown in Example 2.3? For that matter, what is the (row) type of ('S1', 'C1', 'Anne')? In particular, what are the field names of those three fields, which would become column names for the containing table? The short answer is that they are determined by the context in which the expression appears. Because the components are distinguished anyway by ordinal position, the field names sometimes serve little or no purpose. In fact several fields are permitted to acquire the same name. Also, sometimes the context does not provide any names at all, in which case, according to the standard, each field is assigned a unique but unpredictable name. Examples arising as we go along will make this issue a little clearer. I shall use the term *anonymous column* to refer to a column whose name is unpredictable and therefore effectively undefined.

Note carefully that if the word `ROW` is omitted and the row consists of a single field, then the parentheses can also be omitted. Thus, `VALUES 'S1'` denotes a table consisting of a single column and a single row, the SQL counterpart of `RELATION { TUPLE { StudentId 'S1' } }` (though the SQL counterpart has nothing corresponding to the attribute name). Furthermore, recall from Chapter 1, Section 1.15 **Updating Variables** that `VALUES 'S1', 'S2'` denotes a table consisting of a single column and two rows—not a single row and two columns!

The declared type of a `VALUES` expression is determined by that of its contained row expressions and cannot be explicitly stated. As a consequence, SQL has no literals for empty tables—no counterparts of **Tutorial D**'s expressions such as `RELATION { StudentId SID, Name CHAR, CourseId CID } { }`, where heading and body are both specified explicitly.

Effects of NULL

When a `VALUES` expression appears as the source value for an SQL `INSERT` statement, the key word `NULL` can appear as a field value, such that for example `INSERT INTO ENROLMENT VALUES ('S1', NULL, 'C1')` is permitted as short for `INSERT INTO ENROLMENT(StudentId, Name, CourseId) VALUES ('S1', CAST (NULL AS VARCHAR(50)), 'C1')`. (I omitted the column name list in the short form just to remind you of the significance of column order in SQL.)

Historical Note

The history surrounding `VALUES` expressions is described in the historical notes for Section 1.15, **Updating Variables**, in Chapter 1.

2.10 Types and Representations

This section in the theory book introduces the concept *possible representation*, abbreviated *possrep*, and explains how these can be used in conjunction with constraints to define types. Example 2.4 from that book is copied here, along with its explanation.



Hva får egentlig en ingeniør- eller teknologistudent for 300 kroner?

- Medlemskap i en aktiv studentorganisasjon – hele studietiden
- 150 tillitsvalgte studenter som ivaretar dine interesser
- Jobbsøkerkurs
- Gratis PC-forsikring og gode bank- og forsikringstilbud
- Teknisk Ukeblad og NITO Refleks
- Møteplasser på web 2.0

Flere medlemsfordeler og innmelding: www.nito.no/student

Alle som studerer på ingeniør-, bioingeniør-, sivilingeniør eller andre teknologistudier (høgskolekandidat, bachelor eller master) kan bli medlem i NITO.

NITO NORGES STØRSTE ORGANISASJON FOR INGENIØRER OG TEKNOLOGER



Example 2.4: A Type Definition

```

TYPE SID POSSREP SID { C CHAR
                        CONSTRAINT LENGTH(C) <= 5
                        AND
                        STARTS_WITH(C, 'S')
                        AND
                        IS_DIGITS(SUBSTRING(C,1)) )
} ;

```

Explanation 2.4:

- **TYPE SID** announces that a type named `SID` is being defined to the system.
- **POSSREP SID** announces that what follows, in braces, specifies a possible representation for values of the type. It means that the operators defined for type `SID` behave as if values of type `SID` were represented that way, regardless of how they are physically represented “under the covers”. That is why we use the word “possible”—the values might possibly be represented internally that way (but they don’t have to be and we don’t even know if they are). In this case the name of that possrep is the same as the type name, `SID` (as it would be if we omitted the name). (**Tutorial D** allows more than one possrep to be given for the same type, but this feature is beyond the scope of this book. I do not deal with types in any depth. It is sufficient for present purposes just to understand how they exist, what they are for, and how to use them.)
- **C CHAR** defines the first and only component of the possrep, naming it `C` and specifying `CHAR` as its declared type. This definition causes an operator, `THE_C`, to come into existence. `THE_C` takes a value, `s`, of type `SID` and returns the value of the `C` component of `s` under the possible representation `SID`.
- **CONSTRAINT** announces that the expression following it (up to but excluding the closing brace) is a condition that must be satisfied by all possrep values that do indeed represent values of type `SID`. Note that the expression itself uses the logical connective `AND`, with its usual meaning, to connect three expressions, two of which are *comparisons* and each of which is a *truth-valued* expression—one that, when evaluated, yields either `TRUE` or `FALSE`.

The operators `LENGTH`, `STARTS_WITH`, `SUBSTRING`, and `IS_DIGITS`, invoked in the constraint expression, are not defined as built-in operators in **Tutorial D**. I am assuming their existence as user-defined operators. Happily, their definitions are contained in the file `OperatorsChar.d` included in the download package for *Rel*.

- **LENGTH(C) <= 5** expresses a rule to the effect that the total length of a value for the `C` possrep component must never exceed 5.

- **STARTS_WITH(C, 'S')** returns **TRUE** if and only if the string given as the first operand starts with the string given as the second operand—in this case the string consisting of just the capital letter S.
- **SUBSTRING(C, 1)** denotes the string consisting of the whole of the value of the C posrep component apart from the first character. This is given as the argument to an invocation of **IS_DIGITS**, which takes a string and yields **TRUE** if every character in the given string is a numeric digit, otherwise **FALSE**.

Although SQL has more than one way of defining user-defined types, it has no true counterpart of **Tutorial D**'s **TYPE** statement. We will examine three attempts to emulate Example 2.4 in SQL, showing in each case where the attempt falls short. To avoid falling out of synch with the theory book's example numbers, these three are labelled 2.4a, 2.4b, and 2.4c.

Example 2.4a: First attempt at defining type SID in SQL

```
CREATE TYPE SID AS ( C VARCHAR(5) ) ;
```



Skatteetaten



Vil du jobbe i et av landets største IT-miljøer?
Vi skal gjøre det kompliserte enkelt

Skatteetaten tilbyr store fagmiljø og utfordrende oppgaver innen:

- > Systemutvikling
- > Service oriented architecture (SOA)
- > Business intelligence (BI)
- > Testledelse
- > Webutvikling
- > IT sikkerhet
- > Infrastruktur
- > Brukergrensesnitt

For nyutdannede IT-spesialister kan vi tilby et to-årig traineeprogram.

For mer informasjon se skatteetaten.no/jobb

Profesjonell • Nytenkende • Imøtekommende



Explanation 2.4a:

- **TYPE SID** announces that a type named `SID` is being defined to the system.
- **AS (C VARCHAR(5))** defines `SID` as a *structured type*, whose values are represented as a structure consisting in this case of just a single *attribute*, named `C`, of type `VARCHAR(5)`. (The term *attribute* here is not to be confused with its use in relational theory.) The attribute definition `C VARCHAR(5)` causes an operator to come into existence that takes a value, `s`, of type `SID` and returns the value of the `C` component of `s`. The operator, SQL's counterpart of `THE_C`, is invoked using dot notation: `s.C` and is termed the *observer function* for the component `C`.

SQL does not have an immediate counterpart of **Tutorial D's selector**. Instead, a so-called *constructor function*, in this case a niladic operator named `SID`, is generated by the structure definition, such that `SID()` denotes the value of type `SID` whose only component is the “default value” for the attribute `C`, probably `NULL`. To emulate **Tutorial D's SID selector** we need to use the so-called *mutator function* for the attribute `C`, which is also invoked using dot notation: `SID().C('S1')`. The mutator function takes a value of type `SID` as its left operand and a value of the declared type of `C` as its right operand (in parentheses). In general, if `s` is a value of type `SID`, then `s.C('S1')` denotes the `SID` value that is obtained from `s` by replacing its `C` component by the string `'S1'`. If `s` had any other components (it doesn't, of course), they would be retained in `s.C('S1')`. By the way, don't be misled by the term “mutator”: an SQL mutator function is a read-only operator.

Note that although we can restrict the length of the character strings as in Example 2.4, we cannot apply those further constraints concerning the string value. That explains how defining a structured type for student identifiers falls short of emulating Example 2.4.

It is probably unusual in SQL for the structure of a structured type to consist of a single attribute. One would more likely decide to define a *distinct type*, as shown in Example 2.4b.

Example 2.4b: Second attempt at defining type `SID` in SQL

```
CREATE TYPE SID AS VARCHAR(5) ;
```

Explanation 2.4b:

- **TYPE SID** announces that a type named `SID` is being defined to the system.
- **AS VARCHAR(5)** defines `SID` as a *distinct type*, whose values are represented by values of type `VARCHAR(5)`. In this form of type definition the given representation must be a system-defined type.

Under this definition, `SID('S1')` is exactly equivalent to the same expression in **Tutorial D**, and the SQL expression `CAST(s AS VARCHAR(5))`, where `s` is a value of type `SID`, is equivalent to **Tutorial D**'s `THE_C(s)`. However, as in Example 2.4a, we have no way of further constraining the string values representing student identifiers; so `SID('34x.1')`, for example, also denotes a value of type `SID`, as does `SID('')`.

Finally, we could try defining a domain, as in Example 2.4c.

Example 2.4c: Third attempt at defining type `SID` in SQL

```
CREATE DOMAIN SID AS VARCHAR(5)
CHECK ( VALUE IS NOT NULL AND
        SUBSTRING(VALUE FROM 1 FOR 1) = 'S' AND
        CAST('+' || SUBSTRING(VALUE FROM 2) AS INTEGER) >= 0 );
```

Explanation 2.4c:

- **DOMAIN SID** announces that a *domain* named `SID` is being defined to the system.
- **AS VARCHAR(5)** specifies that values in domain `SID` are certain values of type `VARCHAR(5)`.
- **CHECK (...)** specifies a constraint defining exactly which values of type `VARCHAR(5)` are in the domain `SID`. Note that the key word `VALUE`, which is available only in domain constraints, refers (in this particular example) to an arbitrary value of type `VARCHAR(5)`.
- **VALUE IS NOT NULL** specifies that the null value of type `VARCHAR(5)` is not a value in the domain. This is needed because the other conjuncts evaluate to `UNKNOWN` if `VALUE` is the null value and a domain constraint is deemed to be violated only when it evaluates to `FALSE`.
- **SUBSTRING(VALUE FROM 1 FOR 1) = 'S'** specifies that every value in the domain must begin with `S`. Note SQL's deliberate use of “noise” words in the invocation of `SUBSTRING`, the justification for which is to distinguish invocations of system-defined operators from those of user-defined ones.
- **CAST('+' || SUBSTRING(VALUE FROM 2) AS INTEGER) >= 0** is an attempt to emulate an invocation of `IS_DIGITS`, perhaps showing how a user-defined operator of that name might be implemented in SQL. The character “+” is concatenated to the putatively numeric portion of the string in order to exclude values such as `'S+123'` from the domain (the string `'+123'` can be cast as an integer but `'++123'` cannot).

So the domain `SID` gives us the required values, but they are values of the system-defined type `VARCHAR(5)`, not a user-defined type. Thus, the operators defined for these values are exactly those defined for character strings in SQL. Moreover, as SQL does not allow domain names to be given as declared types of parameters or operators, we cannot define any special operators for student identifiers. So defining `SID` as a domain also falls short of emulating Example 2.4.

The syntax for `CREATE TYPE` includes a plethora of optional extras that are not illustrated in Examples 2.4a and 2.4b. Most of them are beyond the scope of this book but it is worth mentioning the `UNDER` clause, whereby a structured type can be specified as a subtype of another structured type. Note first that subtyping is available only with structured types, so in SQL we cannot define, for example, type `POSINT` (positive integers) as a subtype of `INTEGER`. Note also that SQL's model of subtyping is quite different from **Tutorial D's** and is more akin to that found in object-oriented languages, whereby a subtype actually “extends” its supertype by adding extra values (so to speak), rather than being a proper subset of it as specified by a declared constraint. The theory book doesn't have much to say about models of subtyping because relational database theory is generally considered to be independent of type theory, in the sense that it is silent with regard to which types are permitted as declared types of attributes of relations.

Effect of `NULL`

One effect has already been seen in Example 2.4c, where the declared constraint has to explicitly rule out `NULL` from the domain. In Example 2.4a `SID()` denotes the `SID` value whose `C` component is “the null value of type `VARCHAR(5)`”, but note carefully that `SID() IS NULL` evaluates to `FALSE`—it does not denote “the null value of type `SID`”! And yet that null value does exist in SQL, being denoted by `CAST(NULL AS SID)`. By contrast, in Example 2.4b, where `SID` is a distinct type as opposed to being a structured type, `SID(CAST(NULL AS VARCHAR(5)))` is indeed “the null value of type `SID`”.

An Example of Type versus Representation Confusion in SQL

The theory book describes how a value might have two or more distinct representations. For example, user-defined type `POINT` might have a declared `possrep` based on Cartesian coordinates and another based on polar coordinates. SQL has a system-defined type, `TIMESTAMP`, for values representing points in time, a timestamp being represented by a date, a time of day, and—optionally—a time zone, expressed as a displacement from UTC. Clearly any timestamp can be expressed in several different ways, using different time zones. Three o'clock in the afternoon of December 31st, 2011 in UK, for example, is the same time as two o'clock in the morning of January 1st, 2012 in New Zealand.

SQL treats those two representations as distinct values that compare equal, in like fashion to its treatment of character strings with trailing blanks mentioned in Section 2.5 under the heading **Indeterminacy in SQL**, with similar consequences. If instead it had treated them as distinct representations of the same value, then the issue of indeterminacy would not have arisen.

Historical Note

`CREATE DOMAIN` was added to the SQL standard in 1992 but it is still an optional conformance feature and it has not been taken up by many SQL products. Support for user-defined operators arrived in 1996, but nobody in any national body represented on the committee was willing and able to complete the work that had been started on specifications for supporting domain names in the new places where data types are needed, such as parameter definitions, `RETURNS` clauses of operator definitions, and local variable declarations. That's because most substantive change proposals are drafted by experts employed by leading SQL vendors and none of these vendors' products supported domains at the time.

`CREATE TYPE` was added in 1999. Apart from the simple form shown in Example 2.4b it is an optional conformance feature. For the reasons already given, a domain name cannot be used to specify the declared type of an attribute or the representation type of a distinct type.

2.11 What Is a Variable?

SQL's support for variables is very similar to **Tutorial D's**, except that the syntax for creating persistent variables—base tables—is quite different from that used to declare local variables. Example 2.5 is SQL's counterpart of that example in the theory book but, as you know, `CREATE TABLE` is used for base tables.



OLJE- OG ENERGIDEPARTEMENTET



Er du full av energi?

Olje- og energidepartementets hovedoppgave er å tilrettelegge for en samordnet og helhetlig energipolitikk. Vårt overordnede mål er å sikre høy verdiskapning gjennom effektiv og miljøvennlig forvaltning av energiresursene.

Vi vet at den viktigste kilden til læring etter studiene er arbeidssituasjonen. Hos oss får du:

- Innsikt i olje- og energisektoren og dens økende betydning for norsk økonomi
- Utforme fremtidens energipolitikk
- Se det politiske systemet fra innsiden
- Høy kompetanse på et saksfelt, men også et unikt overblikk over den generelle samfunnsutviklingen
- Raskt ansvar for store og utfordrende oppgaver
- Mulighet til å arbeide med internasjonale spørsmål i en næring der Norge er en betydelig aktør

Vi rekrutterer sivil- og samfunnsøkonomer, jurister og samfunnsvitere fra universiteter og høyskoler.

www.regjeringen.no/oed



Se ledige stillinger her

www.jobb.dep.no/oed



Example 2.5: A Variable Declaration

```
DECLARE SN SID DEFAULT SID ( 'S1' ) ;
```

This differs from Example 2.5 in the theory book only in the two key words, `DECLARE` in place of `VAR` and `DEFAULT` in place of `INIT`. The effect is exactly the same apart from the fact that, as already noted, SQL's type `SID` (here assumed to be a distinct type) cannot be the same as **Tutorial D**'s. The key word `DEFAULT` is perhaps a strange choice as that word normally suggests action to be taken by the system when no specific action is explicitly demanded by the user. Here it is used to state explicitly an immediate assignment to the variable being declared.

Example 2.6 in the theory book illustrates the use of a `VAR` statement in **Tutorial D** to create a persistent relation variable and declare a (key) constraint to be satisfied by every relation ever assigned to it. That example is actually a repetition of Example 1.1. For completeness, Example 1.1 of the present book is repeated here as Example 2.6, but slightly revised to make it a truer counterpart of the **Tutorial D** example. The revision is marked in bold.

Example 2.6: Creating a base table.

```
CREATE TABLE ENROLMENT
( StudentId  SID ,
  Name      VARCHAR(30) NOT NULL,
  CourseId  CID ,
  PRIMARY KEY ( StudentId, CourseId )
) ;
```

Explanation 2.6:

- **CREATE TABLE ENROLMENT** announces that what follows defines a variable in the database, named `ENROLMENT`. A variable in an SQL database is necessarily a table variable, just as in a relational database every variable is a relation variable. SQL does not use the term variable, instead referring to the variable as a *base table* (its value being called a table, of course).
- **StudentId SID** defines the first column of `ENROLMENT`, giving its name and either its declared type (a user-defined type) or its domain—we cannot tell which. If `SID` is a domain, then the definition of that domain specifies the declared type of the column `StudentId`. Similarly, `Name VARCHAR(30)` and `CourseId CID` define the second and third columns of `ENROLMENT`, respectively. A system-defined type is explicitly given for the column `Name` but the remarks on the declared type of `StudentId` apply in similar fashion to `CourseId`. Note carefully that in SQL it is correct, in ordinary prose, to identify columns by their ordinal position. By contrast there is no such thing as “the first attribute” of a relation or a relation variable.

- **NOT NULL**, appended to the definition of *Name*, specifies a constraint to the effect that the table assigned to *ENROLMENT* cannot contain a row in which “the null value of type *VARCHAR(30)*” appears for that column. The constraint is needed for accurate emulation of Example 2.6 in the theory book because relational theory does not admit any counterpart of SQL’s *NULL* (so nor does **Tutorial D**). See the next bullet for an explanation of why *NOT NULL* is not appended to the other two column definitions.
- **PRIMARY KEY (StudentId, CourseId)** specifies that at no time can two distinct rows appear in the current value of *ENROLMENT* having the same value for *StudentId* and also the same value for *CourseId*. In enterprise terms, no two enrolments can involve the same student and the same course. In addition, it implies that the *NOT NULL* constraint applies to each those two columns, which explains why those constraints are not explicitly declared in Example 2.6 (though there would be no harm in doing so). Unlike **Tutorial D**, SQL again attaches significance to the order in which the columns of a key are specified. There is more to be said about key constraints in SQL but we defer that to Chapter 6, “Constraints and Updating”. Also unlike **Tutorial D**, an SQL base table whose definition includes no key constraints can exhibit the “duplicate row” phenomenon, allowing its current value to include more than one appearance of the same row.

No initial value for *ENROLMENT* is specified in Example 2.6. SQL does allow one to be specified but does not use the *DEFAULT* syntax of local variable declarations for that purpose. Rather, it allows an initial value to be specified *in place of* the list of column definitions as shown in Example 2.6a—a counterpart of **Tutorial D**’s *VAR* declaration in which the declared type is omitted, being implied by the *INIT* specification. When no initial value is specified, the initial value is the empty table of the specified type.

Effect of Anonymous Columns

Now, recall that a *VALUES* expression denotes a table with undefined column names. If an initial value is to be specified when a base table is created, column definitions have to be implied by that initial value, so the question arises, how can a *VALUES* expression provide the initial value for a base table? The answer is that you have to learn an extra syntactic construct for that purpose, shown in Example 2.6a.

Example 2.6a: Specifying an initial value for a base table.

```
CREATE TABLE ENROLMENT ( StudentId, Name, CourseId )
      AS ( VALUES ( SID('S1'), 'Anne', CID('C1') ),
                  ( SID('S2'), 'Boris', CID('C1') ) )
      WITH DATA ;
```

Explanation 2.6a:

- **(StudentId, Name, CourseId)** provides the names, positionally corresponding to the anonymous columns of the AS table.
- **WITH DATA** specifies that the given table is indeed to be the initial value. Curiously, this is required, unless **WITHOUT DATA** is written instead, in which case the AS table serves only to determine the declared types of the columns of the base table, thus providing something of a counterpart to **Tutorial D's** **SAME_TYPE_AS** construct (though no such facility is available in local variable declarations).
- SQL does not allow constraints to be declared if AS is used in place of explicit column definitions. Example 2.6b shows how the ones given in Example 2.6 could be added subsequently.

Example 2.6b: Adding table constraints later

```
ALTER TABLE ENROLMENT ADD CONSTRAINT NameNotNull  
CHECK ( Name IS NOT NULL ) ;
```

```
ALTER TABLE ENROLMENT ADD CONSTRAINT PK_StudentId_CourseId  
PRIMARY KEY ( StudentId, CourseId ) ;
```



En bok om ting som er greit å vite når du har flyttet hjemmefra.

dnb.no



Bank fra A til Å



Explanation 2.6b:

- **ALTER TABLE ENROLMENT** specifies that the definition of base table `ENROLMENT` is to be modified in some way, such as adding or dropping a column, adding or dropping a constraint (among other things).
- **ADD CONSTRAINT NameNotNull** specifies that in fact a constraint is being added to the definition of `ENROLMENT`, and the name by which that constraint can subsequently be referred to is `NameNotNull`. For example, the name would be needed if the constraint were later to be dropped, or perhaps it could appear in an error message when an attempted update is rejected for violating that constraint. Similar comments apply to `ADD CONSTRAINT PK_StudentId_CourseId`.
- **CHECK (Name IS NOT NULL)** specifies a truth-valued expression, `Name IS NOT NULL`, that must be satisfied by each row of every table that is assigned to `ENROLMENT`. If the current value of `ENROLMENT` fails to satisfy this constraint, then the `ALTER TABLE` statement fails.
- **PRIMARY KEY (StudentId, CourseId)** has the same meaning as in Example 2.6. Again, the `ALTER TABLE` statement fails if the constraint is not satisfied by the current value of `ENROLMENT`.

Note that it is not possible to specify more than one alteration in a single `ALTER TABLE` statement.

Effects of NULL

If *LV* is a local variable and *TLV* is its declared type, then “the null value of type *TLV*” can appear as the value of *LV*, and is indeed its initial value by default.

As already noted in Section 2.8 **The Type of a Table**, there is no null value of any table type, so it is not possible for `NULL` to appear as the value of a base table. Also, `NULL` cannot appear in place of a row in any table. However, if *MRT* is a multiset type whose element type is a row type *RT*, then “the null value of type *MRT*” exists and can appear as the value of a local variable of that type, as the argument substituted for a parameter of that type, as the result of an invocation of an operator of that type, as the value of an attribute of that type in a value of some structured type, and as the value of a field of that type in a row. Moreover, “the null value of type *RT*” can appear as one or more elements of a value of type *MRT*.

Historical Notes

The SQL syntax in Examples 2.6 and 2.6b was defined in SQL:1992, apart from the use of user-defined types. Support for local variables was added in 1996, being included in the definition of the SQL standard's programming language, SQL/PSM. PSM stands for Persistent Stored Modules—user-defined functions and procedures akin to **Tutorial D**'s user-defined read-only operators and update operators, respectively, though actually it defines just the additional language features that are needed to write the implementation code for such modules. There are historical reasons for the inappropriate name for this programming language—the first edition of Part 4, in 1996, included, along with the programming language constructs, the specifications for `CREATE FUNCTION` and `CREATE PROCEDURE` statements, functions and procedures being referred to collectively as “server modules”. When these were later moved to Part 2, SQL/Foundation, the name for Part 4 was not changed.

The `AS` option in Example 2.6a was added in 2003 and is an optional conformance feature. In connection with that, it should be noted that SQL:1992 saw the first appearance of syntax in `CREATE TABLE` whereby some or all of the required column definitions could be copied from an existing base table (or view). For example, `LIKE T2` appearing in place of the i -th column definition for base table `T1` specifies that the definitions of the columns of `T2` are to appear, in order, as the i -th, $i+1$ -th, ... columns of `T1`. Thus, in 2003 the following two statements became equivalent:

```
CREATE TABLE T1 ( LIKE T2 ) ;
CREATE TABLE T1 AS ( SELECT * FROM T2 ) WITHOUT DATA;
```

The `LIKE` option, added in 1992 and still an optional conformance feature, might have had its origins (though I somehow doubt it) in a similar feature of Business System 12, devised in 1978, but BS12's `LIKE` took a table expression of arbitrary complexity rather than being restricted to the name of some existing variable. SQL:1992's `LIKE` was criticised by some (including myself, a member of the standards committee at that time) for being so restrictive, but as we have already seen in the case of `VALUES` expressions, not every table expression in SQL denotes a table whose type meets the special requirements of a base table—in particular, no column of a base table can be anonymous and no two columns can have the same name. This fact would have slightly complicated matters if the restriction on `LIKE` had been removed in 1992, but we have already seen how the problem was addressed when `AS` was added in 2003.

2.12 Updating a Variable

Like **Tutorial D**, SQL supports an explicit assignment operator, illustrated in Example 2.7, but this requires the target to be a local variable, not a base table (or updatable view).

Example 2.7: A Simple Assignment

```
SET SN = SID ( 'S2' ) ;
```

This can obviously be read as “set the variable SN to be equal in value to SID ('S2')”. For completeness we show also SQL’s counterpart of the theory book’s Example 2.8.

Example 2.8: Assignment Source Not a Literal

```
SET SN = SID ( SUBSTRING ( SN.C FROM 1 FOR 1 ) || '5' ) ;
```

Example 2.9 is an exact copy of its counterpart in the theory book.

Example 2.9: Invocation of a user-defined update operator

```
CALL SET_DIGITS ( SN , 23 ) ;
```

Example 2.10 in the theory book illustrates assignment to a “pseudovalue” (that term being taken from the old IBM language PL/I). SQL doesn’t use that term but does have a counterpart of that particular example.

Example 2.10: Assignment of an attribute value in a variable of a structured type

```
SET SN.C = 'S2' ;
```

As in Example 2.10 in the theory book, the entire statement is equivalent to a regular assignment, in this case

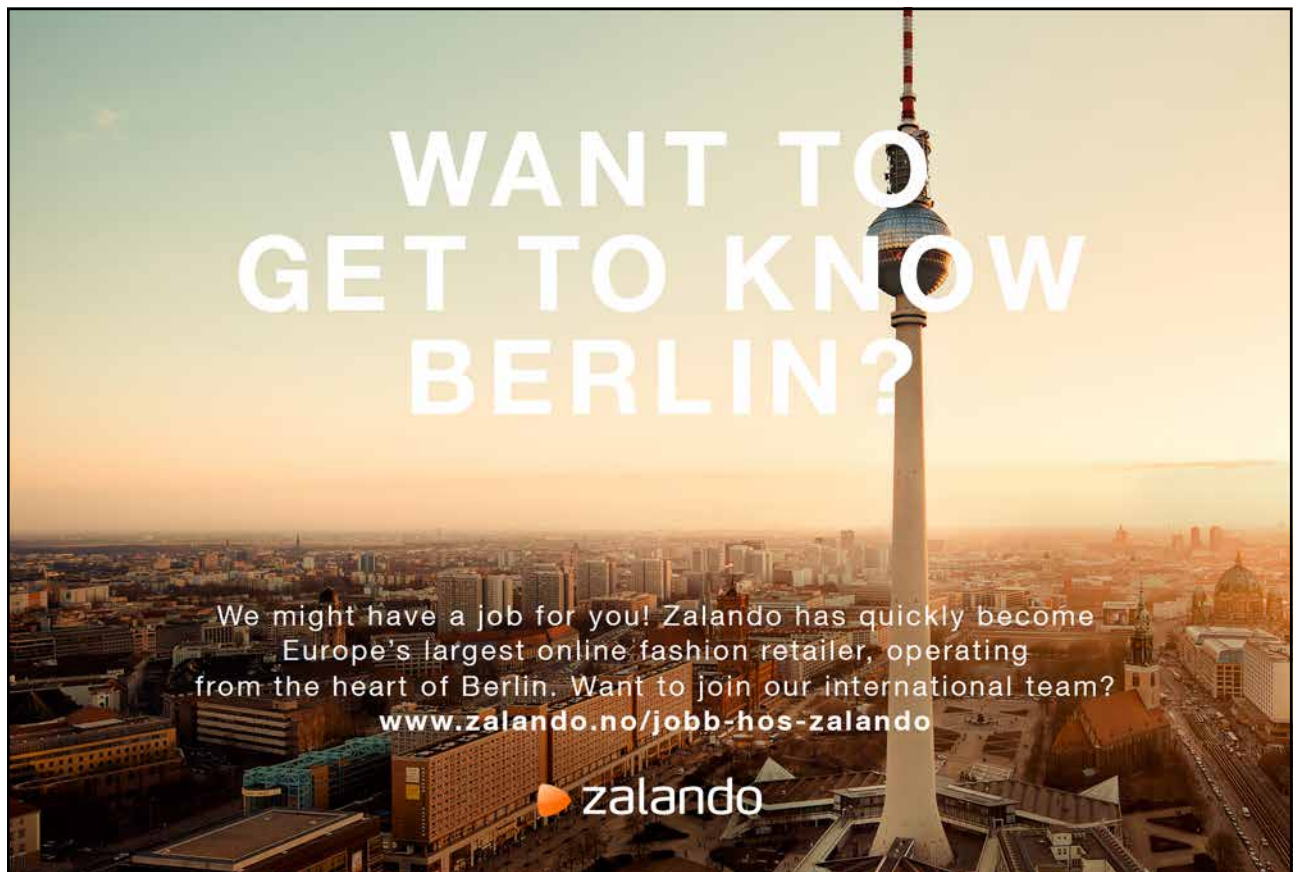
```
SET SN = SID().C('S2') ;
```

—and the remarks in the theory book apply equally well here.

2.13 Conclusion

In the theory book I concluded Chapter 2 by reminding you of the important distinctions I drew to your attention again here in Section 2.3. They are still important, of course, so I repeat them here, with the illustrative examples translated into SQL:

- syntax and semantics (expressions and their meaning)
An expression (syntax) *denotes* either a value or a variable.
- values and variables
A value such as the integer 3, the character string 'London', or the table `VALUES (3, 'London')` is something that exists independently of time or space and is not subject to change. A variable *is* subject to change (in value only), by invocation of update operators. A variable reference, such as the expression `X`, denotes either the variable of that name or its current value, depending on the context in which it appears.
- values and representations of values
The character string value denoted by 'S1' is the *representation* of the student identifier S1, a value of type `SID`, denoted by `SID ('S1')` if `SID` is a distinct type, or by `SID () . C ('S1')` if it is a structured type. (Note: “the representation”, not just “a possible representation” as the theory book has it. SQL doesn’t cater for multiple possible representations.)



- types and representations

For example, if `SID` is a structured type, then `(C VARCHAR (5))` defines the representation for all values of type `SID`. (Note “the” again.)

In connection with type `TIMESTAMP`, SQL’s support for time zone displacements appears to suffer from a failure to distinguish a type from its possible representations, giving rise to needless indeterminacy.

- read-only operators and update operators

“+” is a *read-only operator* because, when it is invoked, it returns a value. “SET” is an *update operator* because, when it is invoked, it has the effect of replacing the current value of a variable (i.e., *updating* the variable by *changing* its value)—and does *not* return a value.

- operators and invocations

`SID` is an *operator*, as it happens, an operator of the same name as the type whose definition brings this operator into existence. Assuming the type `SID` to be a distinct type now, the *signature* of the operator `SID` is `SID (C VARCHAR (5))`. `SID (' S1 ')` denotes an *invocation* of `SID`. Similarly, “+” is an operator, with signature `"+" (A REAL, B REAL)`, and `x+y` denotes an invocation of “+”. (As in **Tutorial D**, “+” has other signatures, defining it for other numeric types such as `INTEGER`.)

- parameters and arguments

`C` is a *parameter* (and in fact the only parameter) of the operator `SID`. `VARCHAR (5)` is the declared type of `C`. Similarly, `x` and `y` denote the arguments substituted for the parameters `A` and `B`, respectively, in the invocation `x+y`.

3 Predicates and Propositions

3.1 Introduction

In Chapter 1 of the theory book I defined a database to be “...an *organised*, machine-readable collection of *symbols*, to be *interpreted* as a *true* account of some *enterprise*.” I also gave this example (extracted from Figure 1.1):

StudentId	Name	CourseId
S1	Anne	C1

I suggested that those green symbols, organised as they are with respect to the blue ones, might be understood to mean:

“Student S1, named Anne, is enrolled on course C1.”

That is how Chapter 3 of the theory book started. It continued with the following paragraph:

In this chapter I explain exactly how such an interpretation can be justified. In fact, I describe the general method under which data organized in the form of relations is to be interpreted—to yield *information*, as some people say. This method of interpretation is firmly based in the science of *logic*. Relational database theory is based very directly on logic. Predicates and propositions are the fundamental concepts that logic deals with.

The remainder of the chapter could be said to apply equally well to the interpretation of SQL databases were it not for SQL's use of a logic based on three truth values instead of the usual two, this arising from its special construct referred to as `NULL`. Many of the effects of this intrusion have already been examined in Chapter 2. Here I make further observations that arise in connection with the corresponding sections of this chapter in the theory book.

3.2 What Is a Predicate?

Consider the declarative sentence—a proposition—that is used to introduce this topic in the theory book:

“Student S1, named Anne, is enrolled on course C1.”

Recall that the terms S1, Anne, and C1 are *designators*, each referring unambiguously to a particular thing. The chapter later explains how a tuple can provide values—*attribute* values—to be interpreted as designators to be substituted for the corresponding parameters of a predicate. Thus, this sentence might be represented by the tuple denoted in **Tutorial D** by `TUPLE{StudentId SID('S1'), Name NAME('Anne'), CourseId CID('C1')}`. As SQL allows NULL to appear wherever a value can appear, we have to entertain the notion that the row denoted in SQL by `(SID('C1'), NULL, CID('C1'))` might represent some sentence. (*Aside:* SQL does not use attribute names to connect values to their corresponding parameters. Instead, the correspondence is determined by position and I have assumed that the parameters are to be considered in the order *StudentId*, *Name*, *CourseId*. *End of aside.*) Now, could that sentence be “Student S1, named NULL, is enrolled on course C1”? Well, no, because NULL is not a name and really doesn’t designate anything. The row might instead represent the sentence “Student S1, whose name is not known, is enrolled on course C1”. But that sentence contains nothing that can be regarded as a designator substituted for the parameter *Name*.

If we now recast this into two simpler sentences, as in the theory book, we will get something like, “Student S1’s name is not known” and “Student S1 is enrolled on course C1”. Let’s now try replacing S1 by NULL in the second of those:

Example 3.1:

“Student NULL is enrolled on course C1.”

Again that doesn’t make sense. If NULL is *always* to be interpreted as meaning “some value should appear here but we don’t know which”, then perhaps the sentence should be “Some student, whose student identifier is not known, is enrolled on course C1.” But again, that sentence contains nothing that can be regarded as a designator substituted for the parameter *StudentId*.

Similarly, NULL, might appear in place of C1:

Example 3.2:

“Student S1 is enrolled on course NULL.”

and again we have to reject that and write instead, perhaps, “Student S1 is enrolled on some course, whose course identifier is not known”, or perhaps, “Student S1 is enrolled on some course but we don’t know which one”. The row $(SID('S1'), NULL)$ could indeed mean either of those and either would be consistent with the notion of “some value appears here but we don’t know which”. However, in certain operations, such as the “outer joins” that we shall meet in Chapter 4, SQL uses that very row to mean “Student S1 is not enrolled on any courses”, which, although perhaps a more likely interpretation in practice, is *not* consistent with the meaning “some value appears here but we don’t know which” (nor with the fact that SQL would not regard two such students as being enrolled on the same set of courses).

3.3 Substitution and Instantiation

Section 3.2 shows how `NULL` might appear in substitution for a parameter of a predicate and how it might thus participate in instantiation of that predicate to yield a proposition. Now consider instantiations of the dyadic predicate $a < b$. As well as instantiations such as $5 < 10$ (a *true* one) and $9 < 6$ (a *false* one), we now have to entertain the possibility of instantiations such as $5 < NULL$, $NULL < 6$, and $NULL < NULL$. In SQL these comparisons evaluate to that intrusive truth value, *unknown*. Now, Section 3.2 in the theory book goes on to explain that the *extension* of a predicate consists exactly of those instantiations of it that evaluate to *true*, from which we can conclude, of every instantiation that does not appear in the extension, that it is *false*, in which case it must appear instead in the extension of the negation of that predicate. In SQL, then, the instantiation $5 < NULL$, for example, cannot be considered to appear in either the extension of $a < b$ or $NOT(a < b)$. Or so it would appear.

“I studied English for 16 years but...
...I finally learned to speak it in just six lessons”
Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



3.4 How a Table Represents an Extension...

...or does it? The theory book here describes how each tuple in a relation represents a true instantiation of some predicate and each true instantiation is represented by some tuple in that relation. Thus, a relation represents an extension, its body containing exactly one tuple corresponding to each element of the extension.

It is true that some SQL tables can be interpreted in this way but it is also true that some SQL tables cannot. In fact there are at least two distinct ways in which an SQL table cannot be thus interpreted:

- a) In SQL it is possible for the same row to appear more than once in a table. Moreover, if tables $t1$ and $t2$ differ only in the number of appearances of some row, then that difference is significant—they are not the same table.
- b) Although I have noted that in SQL the instantiation $5 < \text{NULL}$ cannot be considered to appear in either the extension of $a < b$ or $\text{NOT}(a < b)$, the row $(5, \text{NULL})$ *can* appear in a table. What could be the corresponding predicate? It would have to be some dyadic predicate, $P(a, b)$ say, such that $P(5, \text{NULL})$ is *true*. But if NULL stands for “some value but we don’t know which”, how could that row appear in the same table as, say, $(6, 12)$? If $(6, 12)$ means “6 is related to 12” then $(5, \text{NULL})$, in relational theory, would have to mean that 5 is related to NULL in that same way. But it can’t, because NULL doesn’t designate anything. If instead it means “5 is related to something whose identity is unknown”, then we have a sentence in which nothing appears in substitution for the parameter b .

3.5 Deriving Predicates from Predicates

The corresponding section in the theory book describes how predicates can be derived from predicates using (a) the logical connectives of the propositional calculus, such as AND, OR, and NOT, and (b) quantifiers, such as “there exists” (\exists) and “for all” (\forall). Here I examine how SQL’s truth value, *unknown*, intrudes on those connectives and quantifiers.

Logical Connectives

For these I give SQL’s extended truth tables in which the symbol \cup , for *unknown*, appears along with the usual T and F.

Negation (NOT , \neg)

p	$\neg p$
T	F
U	U
F	T

Figure 3.1: The SQL Truth Table for Negation

We now have three rows instead of just two. As you can see, $\neg p$ is defined as in two-valued logic (2VL) when p is either *true* or *false*, but $\neg(\text{unknown})$ is *unknown*.

Other monadics

In 2VL there are just 4 (2^2) monadic operators, of which negation is really the only “useful” one. When a third truth value is introduced we have 27 (3^3) monadics and SQL gives names to several of these in addition to NOT for its version of negation. Some of these are shown in Figure 3.1a.

p	p IS TRUE	p IS UNKNOWN	p IS FALSE	$p = \text{TRUE}$	$p = \text{UNKNOWN}$	$p = \text{FALSE}$
T	T	F	F	T	U	F
U	F	T	F	U	U	U
F	F	F	T	F	U	T

Figure 3.1a: Truth Tables for Some Other SQL Monadics

Note that under none of the “IS” operators shown in Figure 3.1a does a truth value in the first column map to *unknown*—contrast this with the treatment of “=”. In addition to those three SQL also has p IS NOT TRUE, p IS NOT UNKNOWN, and p IS NOT FALSE, equivalent to $\text{NOT}(p \text{ IS TRUE})$, $\text{NOT}(p \text{ IS UNKNOWN})$, and $\text{NOT}(p \text{ IS FALSE})$, respectively. The test $x = x \text{ IS UNKNOWN}$ can be useful in cases where it evaluates to TRUE when $x \text{ IS NULL}$ does not—for examples, see the **Effects of NULL** in Chapter 2, Section 2.10, **Types and Representations**.

We turn now to the dyadic operators, noting that with three truth values there are now 19,683 (3 to the power 3^2) all told, compared with just 16 (2 to the power 2^2) in 2VL. SQL directly supports (i.e., has names for) just eight of these, including counterparts of conjunction, disjunction, and—surprisingly—implication (which, as we shall see, appears to have been included in the language by accident).

Conjunction (AND, \wedge)

p	q	$p \wedge q$
T	T	T
T	U	U
T	F	F
U	T	U
U	U	U
U	F	F
F	T	F
F	U	F
F	F	F


Figure 3.2: The SQL Truth Table for AND

Now we have nine rows (3^2) instead of just four (2^2). Again, when *unknown* is not involved, the rows are as for 2VL. Also, when anything is paired with *false*, the result is *false*, as in 2VL. Our intuition, that “ p and q ” is true exactly when both operands are true, is preserved.

Disjunction (OR, \vee)

p	q	$p \vee q$
T	T	T
T	U	T
T	F	T
U	T	T
U	U	U
U	F	U
F	T	T
F	U	U
F	F	F

Figure 3.3: The SQL Truth Table for Disjunction



WHILE YOU WERE SLEEPING...

www.fuqua.duke.edu/whileyouweresleeping

DUKE
THE FUQUA
SCHOOL
OF BUSINESS



Again we have nine rows instead of just four and again, when *unknown* is not involved, the rows are as for 2VL. Also, when anything is paired with *true*, the result is *true*, as in 2VL. Our intuition, that “*p* or *q*” is true exactly when at least one operand is true, is preserved.

Now, in the theory book it is noted that disjunction could equally well be defined in terms of conjunction and negation, as

$$p \vee q \equiv \neg(\neg p \wedge \neg q)$$

and the truth table in Figure 3.4 of that book is given as proof of that equivalence. The question arises, does the same equivalence hold in SQL? To answer that we need to look at the revised Figure 3.4.

p	q	$\neg p$	$\neg q$	$\neg p \wedge \neg q$	$\neg(\neg p \wedge \neg q)$
T	T	F	F	F	T
T	U	F	U	F	T
T	F	F	T	F	T
U	T	U	F	F	T
U	U	U	U	U	U
U	F	U	T	U	U
F	T	T	F	F	T
F	U	T	U	U	U
F	F	T	T	T	F

Figure 3.4: SQL Disjunction in Terms of SQL Negation and SQL Conjunction

As you can see, the final column is the same as in Figure 3.3, so that equivalence does also hold in SQL.

Conditionals

At first sight SQL does not appear to have a single operator for expressing logical implication. In this respect it would be in common with most programming languages, including **Tutorial D**. However, standard SQL defines a partial ordering for its three truth values, under which *false* is deemed to precede *true*. Thus, the comparisons $p < q$, $p > q$, $p \leq q$, and $p \geq q$ are all supported in standard SQL (in addition to $p = q$, of course).

Now, in Section 3.5 of the theory book it is noted that in 2VL $p \rightarrow q$ is equivalent to $\neg p \vee q$. Study of Figure 3.5 reveals that $\neg p \vee q$ does indeed equate to $p \rightarrow q$ when neither operand is *unknown*, and the same is true of $p \leq q$! (It is the pronunciation, “is less than or equal to”, rather than “implies”, that led to my observation that SQL appears to include direct support for a 3VL form of implication *by accident*.)

p	q	$\neg p$	$\neg p \vee q$	$p \leq q$
T	T	F	T	T
T	U	F	U	U
T	F	F	F	F
U	T	U	T	U
U	U	U	U	U
U	F	U	U	U
F	T	T	T	T
U	U	U	U	U
F	F	T	T	T

Figure 3.5: The SQL Truth Tables for $\neg p \vee q$ and $p \leq q$

Note, however, that $p \leq q$ is not equivalent to $\neg p \vee q$. Intuitively, we understand that “ p implies q ” is true whenever q is true. This holds for $\neg p \vee q$ but not for $p \leq q$, as the row for $p = \text{U}$ and $q = \text{T}$ shows. The **U** in the last column for that row arises from SQL’s general rule that whenever an operand of a comparison is **NULL**, the result is *unknown*—and **NULL**, when it is the result of evaluating a Boolean expression, is considered synonymous with *unknown*. In fact, Figure 3.5 gives a demonstration of the fact that SQL is not always faithful to its own concept, that **NULL** represents “a value exists here but we don’t know which value”. What **U** really means when it appears in the column for $p \leq q$ is that \leq is *undefined* for that particular pair of truth values.

The biconditional $p \leftrightarrow q$ can be expressed in **Tutorial D** by $p = q$ and the same is true of SQL. The question then arises as to whether, in SQL, $p = q$ is equivalent to $(\neg p \vee q) \wedge (\neg q \vee p)$. This matter is investigated in the truth table of Figure 3.6.

p	q	$\neg p \vee q$	$\neg q \vee p$	$(\neg p \vee q) \wedge (\neg q \vee p)$	$p = q$
T	T	T	T	T	T
T	U	U	T	U	U
T	F	F	T	F	F
U	T	T	U	U	U
U	U	U	U	U	U
U	F	U	T	U	U
F	T	T	F	F	F
F	U	T	U	U	U
F	F	T	T	T	T

Figure 3.6: SQL $p = q \equiv (\neg p \vee q) \wedge (\neg q \vee p)$

As you can see, the equivalence does hold in SQL, but only because SQL treats *unknown* as not equal to—i.e., not the same truth value as—itself! This treatment of $p = q$ is consistent with the general rule that applies when NULL is an operand of a comparison in SQL.

Figure 3.6a similarly investigates whether $p = q$ is equivalent to $(p \leq q) \wedge (q \leq p)$, and as you can see, the equivalence again holds in SQL.

p	q	$p \leq q$	$q \leq p$	$(p \leq q) \wedge (q \leq p)$	$p = q$
T	T	T	T	T	T
T	U	U	U	U	U
T	F	F	T	F	F
U	T	U	U	U	U
U	U	U	U	U	U
U	F	U	U	U	U
F	T	T	F	F	F
F	U	U	U	U	U
F	F	T	T	T	T

Figure 3.6a: SQL $p = q \equiv (p \leq q) \wedge (q \leq p)$

Quantification

To quantify something, as the theory book has it, is to state its quantity, to say how many of it there are. For example, in **Tutorial D** the expression $\text{COUNT}(r)$ denotes the number of tuples in the relation r , to be interpreted as the number of objects represented by those tuples that satisfy a predicate that r is considered to represent. Universal quantification—stating that something is true of all objects under consideration—is involved in expressions such as

- $\text{AND}(r, c)$, meaning that all objects that satisfy a predicate for r also satisfy the condition (another predicate) c , and
- $\text{IS_EMPTY}(r)$, meaning that no object satisfies a predicate for r —in other words, every object satisfies the negation of that predicate.

Existential quantification—stating that something is true of at least one object under consideration—can be expressed by $\text{OR}(r, c)$, meaning that at least one object that satisfies a predicate for r also satisfies c , and $\text{IS_NOT_EMPTY}(r)$.

The names for the aggregate operators AND and OR reflect the facts that when we confine our attention to finite sets, universal and existential quantification are equivalent to repeated invocations of dyadic AND and dyadic OR , respectively. Note that $\text{AND}(r, c)$ is equivalent to $\text{COUNT}(r) = \text{COUNT}(r \text{ WHERE } c)$, and $\text{OR}(r, c)$ is equivalent to $\text{COUNT}(r \text{ WHERE } c) > 0$ and also to $\text{IS_NOT_EMPTY}(r \text{ WHERE } c)$.



Vi vokser i Norge
og har virksomhet
helt frem til 2050

Er du interessert i sommerjobb
eller fast stilling?

Se informasjon om sommerjobber på
www.bp.no

bp



Quantification also appears in various guises in SQL, but its meaning is muddled by those same two violations of relational theory that we have already seen muddying the waters: duplicate rows and NULL. For example, SQL's `(SELECT COUNT(*) FROM r)`, a so-called scalar subquery (because it is an expression denoting a table with one row and one column, enclosed in parentheses), denotes the number of rows in the table r , but can we really say that this represents the number of objects that satisfy a predicate for r , if the same row can be counted more than once, or if NULL appears in place of a column value in some row of r ? In fact, what might it mean to say that a row does or does not *satisfy* a predicate? In 2VL we say that object a satisfies predicate $P(x)$ exactly when $P(a)$ is *true*. Does this still hold in 3VL, or might SQL deem a to satisfy $P(x)$ also when $P(a)$ is *unknown*? Well, it turns out that SQL uses both interpretations, depending on the context, as we shall discover.

SQL counterparts of **Tutorial D** quantifications

Consider **Tutorial D**'s `AND (r, c)`, where r is a relation and c is a condition that is applicable to tuples of r . This variety of AND is an aggregate operator in **Tutorial D**. The expression evaluates to *true* when every tuple in r satisfies c , otherwise *false*. Looking for an SQL counterpart of this expression, we are faced with a plethora of possibilities. It is a salutary exercise to examine the apparent choices to see which, if any, is the best fit. For aggregation SQL provides what it calls "aggregate functions". These are counterparts, not to **Tutorial D**'s aggregate operators but rather to its constructs such as `SUM(x)` that can appear in invocations of SUMMARIZE. Aggregate functions are used in several of the candidates we shall examine. It is important to bear in mind two general rules that apply to aggregation in SQL. The first is that appearances of NULL are always excluded, such that, for example `SUM(x)` evaluates to 3 when summing 1, 2 and NULL, even though `1+2+CAST(NULL AS INTEGER)` evaluates to NULL. The second is that, except in the case of COUNT, aggregation over the empty set always yields NULL, even though the sum of no integers, for example, really should be zero and the AND of no truth values should be TRUE. With that in mind, let us now look at some of the possibilities for determining whether condition c is satisfied by every row of table r .

1. `(SELECT EVERY(c) FROM r)`

The result is *false* if c evaluates to *false* for at least one row of r , *unknown* if c evaluates to *unknown* for each row of r (including the case where r is empty), otherwise *true*. Note that the treatment thus differs from `AND (r, c)` when r is empty, the result being *unknown* instead of *true*. Note also that the result can be *true* even when c does not evaluate to *true* for every row of r , namely, when c is *true* for at least one row and *unknown* for each of the others. Here, then, we can observe that a row appears to satisfy c if c evaluates to either *true* or *unknown*.

2. `(SELECT MIN(c) FROM r)`

This is available as a result of the partial ordering of truth values previously mentioned. It is equivalent to `(SELECT EVERY(c) FROM r)`.

3. `(SELECT EVERY (c IS TRUE) FROM r)`

Here we are explicitly stating that we deem r to satisfy c only when c is *true* for r . But it is still the case that the expression yields *unknown* when r is empty.

4. `TRUE =ALL (SELECT c FROM r)`

Here we are using the unusual construct SQL calls a “quantified comparison predicate”. For example, the comparison $X =ALL (SELECT Y FROM T)$ results in *true* if $X = Y$ is *true* for every row of r (including the case where r is empty), *false* if $X = Y$ is *false* for at least one row of r , otherwise *unknown*. (Note that the word `ALL` is attached to the comparison operator, not the table expression that follows it. `ALL (SELECT Y FROM T)` is not a legal expression in SQL.) So `TRUE =ALL (SELECT c FROM r)` yields the same result as `AND (r, c)` exactly when c is *true* for every row of table r , but otherwise it can yield either *unknown* or *false*.

5. `TRUE =ALL (SELECT c IS TRUE FROM r)`

At last we have an expression that yields the same result as `AND (r, c)` when c is *true* for every row of table r and otherwise yields *false*.

We can conduct a similar investigation in connection with **Tutorial D**’s `OR (r, c)`:

1. `(SELECT SOME (c) FROM r)`

The result is *true* if c evaluates to *true* for at least one row of r , *unknown* if c evaluates to *unknown* for each row of r (including the case where r is empty), otherwise *false*. The treatment differs from `OR (r, c)` in the case where r is empty and the result is *unknown*—you might find that a bit strange when you consider that if r contains no rows, then obviously there doesn’t exist a row in r that satisfies c .

2. `(SELECT MAX (c) FROM r)` and `(SELECT ANY (c) FROM r)`

These are both equivalent to `(SELECT SOME (c) FROM r)`. `ANY` is just an alternative spelling for `SOME`.

3. `(SELECT SOME (c IS TRUE) FROM r)`

This is also equivalent to `(SELECT SOME (c) FROM r)`. The addition of `IS TRUE` has no effect this time.

4. `TRUE =SOME (SELECT c FROM r)`

This differs from `(SELECT SOME (c) FROM r)` because it yields *false* instead of *unknown* when r is empty and also because it yields *unknown* when c evaluates to *false* for at least one row of r and *unknown* for all the others.

5. `TRUE =SOME (SELECT c IS TRUE FROM r)`

Similar to our fifth candidate for `AND(r, c)`, we finally have an expression that yields the same result as `OR(r, c)` when c is *true* for some row of table r and otherwise yields *false*.

For **Tutorial D**'s `IS_EMPTY(r)` SQL has `NOT EXISTS(r)`, which yields *true* whenever r is empty, otherwise *false*. Note, then, that `NOT EXISTS(r WHERE FALSE)` is not equivalent to `(SELECT EVERY(FALSE) FROM r)`, because of the difference in treatment of the empty table.

Similarly, for `IS_NOT_EMPTY(r)` SQL has `EXISTS(r)`, which yields *False* whenever r is empty, otherwise *True*. Note, then, that `EXISTS(r WHERE TRUE)` is not equivalent to `(SELECT SOME(TRUE) FROM r)`. Note also that `EXISTS(r WHERE c)` evaluates to *false* in the case where r is not empty, c evaluates to *unknown* for at least one row of r , and c evaluates to *false* for every other row. Thus, although SQL uses the name `EXISTS` for this operator, it is not the 3VL existential quantifier. Similarly, `NOT EXISTS(r WHERE NOT (c))` does not in general express universal quantification in 3VL.



Historical Notes

It is commonly believed that the term Structured Query Language, sometimes taken to be the full name for SQL, is inspired by the SELECT-FROM-WHERE structure. This may be the case, but it is not clear whether that was the intention of the authors of SEQUEL. The Abstract for that paper gives a clue: “Moreover, the SEQUEL user is able to compose these basic templates [SELECT-FROM-WHERE templates] in a structured manner to form more complex queries.” That “structured manner” might have referred to SEQUEL’s support for nesting one SELECT-FROM-WHERE structure within another.

The syntax `SELECT * FROM` was not included in SEQUEL because the `SELECT` clause itself was optional, as was the key word `FROM`. Thus, SQL expressions such as `SELECT * FROM T1` and `SELECT * FROM T1, T2` could be written as just `T1` and `T1, T2` in SEQUEL. The shorthand `TABLE t` was added to the SQL standard in 1992 but remains an optional conformance feature.

The monadic operators `IS TRUE`, `IS FALSE`, `IS UNKNOWN` and their negated counterparts `IS NOT TRUE`, `IS NOT FALSE`, `IS NOT UNKNOWN` were added to the language in SQL:1992. They remain optional conformance features.

Support for comparison operators on values of type `BOOLEAN`, along with the aggregate functions `EVERY`, `SOME`, `ANY`, and `MAX` and `MIN` on Booleans, arrived in SQL in the 1999 edition of the international standard, as already noted in Chapter 2. The partial ordering of truth values was perhaps partly a consequence of SQL’s treatment of `NULL` when the rows of a table are to be placed in some specified order (typically by use of an `ORDER BY` clause). For example, suppose that rows of `ENROLMENT` are to be placed in alphabetical order of `NAME`, for which `NULL` appears in some row. Does this row appear before the rows for Anne or after the rows for Zack? When the first edition of the SQL standard (1986) was being drafted it was discovered that existing implementations were divided fairly evenly between those that placed `NULL` first in the ordering and those that placed it last. Rather than toss a coin to decide which implementations would be deemed in conformance, the committee decided not to legislate on this matter. When `BOOLEAN` was added in 1999, the treatment of comparisons on values of this type was at least consistent with that decision. It was also consistent with the existing treatment of comparisons on values of all other types.

As an aside, it is interesting to observe that SQL:2003 included some new material in connection with `ORDER BY`, allowing the *user* to specify the treatment of `NULL`, by writing either `NULLS FIRST` or `NULLS LAST`. However, no similar addition appears in connection with comparisons.

4 Relational Algebra— The Foundation

4.1 Introduction

The theory book's Chapter 4 describes some operators, as manifested in **Tutorial D**, that together constitute an algebra that is not only *relationally complete* but also irreducibly so (very nearly—apart from `RENAME`, which can be expressed in terms of extension and projection, none of those operators can be discarded without sacrificing completeness). We can use these operators as a basis for testing SQL for relational completeness. If we can show that for every invocation of one of these **Tutorial D** operators there is an equivalent SQL expression, then we will have shown that SQL is relationally complete. By “equivalent” we mean an expression whose table operands are counterparts of the **Tutorial D** relation operands (ignoring the ordering that SQL imposes on the columns) and whose result is a table counterpart of **Tutorial D**'s result, where a table is a counterpart of a relation if and only if it satisfies all of the following conditions:

- every column has a name
- no two distinct columns have the same name
- no row appears more than once
- `NULL` doesn't appear in place of a value anywhere in the table
- every row consists entirely of its column values and doesn't somehow contain any additional data

Strictly speaking, SQL cannot be regarded as relationally complete until it recognizes the existence of relations of degree zero: `TABLE_DEE` and `TABLE_DUM` in **Tutorial D**. Charitably overlooking that omission, we will discover that SQL is otherwise relationally complete, though it wasn't so prior to 1992. However, the table counterparts we will find in SQL will give opportunities for further comment on the language design. We will also discover some of the consequences that arise when SQL's operators are used on tables that do not satisfy all of these conditions.

Figure 4.1, repeated from the theory book, shows the current values of relvars named `IS_CALLED` and `IS_ENROLLED_ON`, which we will now take to be SQL tables (as the current values of SQL base tables).

Please note very carefully that all examples and accompanying explanations assume, unless otherwise stated to the contrary, that the tables involved satisfy the above conditions. Otherwise, as they say, “all bets are off”.

IS_CALLED	
StudentId	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

IS_ENROLLED_ON	
StudentId	CourseId
S1	C1
S1	C2
S2	C1
S3	C3
S4	C1

Figure 4.1: IS_CALLED and IS_ENROLLED_ON

This will be our running example here too.

In the theory book Example 4.1 is the first example of an invocation of a relational operator, using JOIN to derive the ENROLMENT relation from IS_CALLED and IS_ENROLLED_ON. Here it shows an SQL equivalent to `IS_CALLED JOIN IS_ENROLLED_ON`.

Example 4.1: Joining IS_CALLED and IS_ENROLLED_ON in SQL

```
SELECT * FROM IS_CALLED NATURAL JOIN IS_ENROLLED_ON
```

gaiteye
Challenge the way we run

EXPERIENCE THE POWER OF
FULL ENGAGEMENT...

.....

RUN FASTER.
RUN LONGER..
RUN EASIER...

READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM



This is an example of an SQL *table expression*. I have been assuming you are already familiar with the SELECT-FROM-WHERE structure for certain table expressions. Here I give an explanation in a different style from that found in the SQL text books, appealing to the concept of operator invocation that is used in the theory book.

Explanation 4.1

- Example 4.1 is an invocation of the SQL operator `SELECT` and for that reason I shall refer to such table expressions as `SELECT` expressions.
- Here the `SELECT` operator operates on the table denoted by the table expression `FROM IS_CALLED NATURAL JOIN IS_ENROLLED_ON`, an invocation of the operator `FROM`. I shall call such table expressions `FROM` expressions.
- The `FROM` operator here is operating on the table denoted by the table expression `IS_CALLED NATURAL JOIN IS_ENROLLED_ON`, an invocation of the operator `NATURAL JOIN`.
- `NATURAL JOIN` here is operating on the tables denoted by the table expressions `IS_CALLED` and `IS_ENROLLED_ON`, each in turn denoting the table that is the current value of the variable (base table) of that name.
- `NATURAL JOIN` is almost equivalent to **Tutorial D**'s `JOIN`. It differs only in being noncommutative because of the ordering to the columns of an SQL table. The common columns appear first in the result, in the order in which they appear in the left operand. Then come the remaining columns of the left operand, followed by the remaining columns of the right operand. As in **Tutorial D**, common columns must be of the same type in both operands.
- `FROM` is an operator that takes a commalist of one or more table expressions. In this example the list has just one element, `IS_CALLED NATURAL JOIN IS_ENROLLED_ON`, and the result is that table. An invocation of `FROM` is usually referred to as a `FROM` clause. A `FROM` clause is not permitted to exist in isolation—it must appear in some containing `SELECT` expression. Similarly, some table expressions are permitted only when they appear as elements of a `FROM` clause. Simple table names and invocations of `NATURAL JOIN` are a case in point. The result of a `FROM` clause must always be operated on by some other clause. In Example 4.1 it is operated on by a `SELECT` clause. It can also be operated on by any clause that immediately follows it syntactically, such as a `WHERE` clause, for example. As we shall see, SQL dictates a strict order in which the clauses of a `SELECT` expression must appear. Evaluation always starts at the `FROM` clause, then proceeds forwards from clause to clause, then finally back to the `SELECT` clause.

- `SELECT` is an operator that takes an explicit or, as in Example 4.1, implicit commalist of column specifications followed by an invocation of `FROM`. The text from the word `SELECT` up to, but not including, the word `FROM` is usually referred to as a `SELECT` clause. The clauses following the `SELECT` clause define a table on which the `SELECT` clause operates. The shorthand `SELECT *` is equivalent to a commalist specifying each column in turn of the input table defined by the following clauses. Thus, `SELECT * FROM t` is very similar to **Tutorial D**’s identity projection, $t\{\text{ALL BUT}\}$ —it yields the table t . (When t is just a single table name, the shorthand `TABLE t` is available as equivalent to the `SELECT` expression `SELECT * FROM t`.)

Historical Note:

It is commonly believed that the term Structured Query Language, sometimes taken to be the full name for SQL, is inspired by the `SELECT-FROM-WHERE` structure. This may be the case, but it is not clear whether that was the intention of the authors of SEQUEL. The Abstract for that paper gives a clue: “Moreover, the SEQUEL user is able to compose these basic templates [`SELECT-FROM-WHERE` templates] in a structured manner to form more complex queries.” That “structured manner” might have referred to SEQUEL’s support for nesting one `SELECT-FROM-WHERE` structure within another.

The syntax `SELECT * FROM` was not included in SEQUEL because the `SELECT` clause itself was optional, as was the key word `FROM`. Thus, SQL expressions such as `SELECT * FROM T1` and `SELECT * FROM T1, T2` could be written as just `T1` and `T1, T2` in SEQUEL. The shorthand `TABLE t` was added to the SQL standard in 1992 but remains an optional conformance feature.

Figure 4.2 shows the result of evaluating Example 4.1. You can see that it depicts exactly the same table as the current value of `ENROLMENT` shown in Figure 1.2. Note in particular that the left-to-right order of the columns is as shown in Figure 1.2. When the table operands are reversed the result is the different SQL table depicted in Figure 4.2a.

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 4.2: The result of `IS_CALLED NATURAL JOIN IS_ENROLLED_ON`

StudentId	CourseId	Name
S1	C1	Anne
S1	C2	Anne
S2	C1	Boris
S3	C3	Cindy
S4	C1	Devinder

Figure 4.2a: The result of IS_ENROLLED_ON NATURAL JOIN IS_CALLED

Effect of NULL

Let c be a common column in $t1 \text{ NATURAL JOIN } t2$. Then there can be no row in the result for which $c \text{ IS NULL}$ evaluates to TRUE, even if both operands contain such a row. In fact, for each common column c , $c=c \text{ IS UNKNOWN}$ must evaluate to FALSE for every row of the result. (Recall that $c=c \text{ IS UNKNOWN}$ can evaluate to TRUE even when $c \text{ IS NULL}$ does not.)

Historical Notes

It is commonly believed that the term Structured Query Language, sometimes taken to be the full name for SQL, is inspired by the SELECT-FROM-WHERE structure. This may be the case, but it is not clear whether that was the intention of the authors of SEQUEL. The Abstract for that paper gives a clue: “Moreover, the SEQUEL user is able to compose these basic templates [SELECT-FROM-WHERE templates] in a structured manner to form more complex queries.” That “structured manner” might have referred to SEQUEL’s support for nesting one SELECT-FROM-WHERE structure within another.

The syntax `SELECT * FROM` was not included in SEQUEL because the `SELECT` clause itself was optional, as was the key word `FROM`. Thus, SQL expressions such as `SELECT * FROM T1` and `SELECT * FROM T1, T2` could be written as just `T1` and `T1, T2` in SEQUEL. The shorthand `TABLE t` was added to the SQL standard in 1992 but remains an optional conformance feature.

What if NATURAL JOIN is missing?

In the absence of `NATURAL JOIN` Example 4.1 has to be replaced by something rather more longwinded, as shown in Example 4.1a.

Example 4.1a: Joining IS_CALLED and IS_ENROLLED_ON in original SQL

```
SELECT IC.StudentId, Name, CourseId
FROM   IS_CALLED AS IC, IS_ENROLLED_ON AS IE
WHERE  IC.StudentId = IE.StudentId
```

Explanation 4.1a

- The `FROM` clause now has two elements. When there are two elements, $t1$ and $t2$, the result is equivalent to $t1 \text{ CROSS JOIN } t2$, which is SQL's counterpart of $t1 \text{ TIMES } t2$ in **Tutorial D**. However, `TIMES` requires its operands to have disjoint headings, whereas `CROSS JOIN` is defined for all pairs of SQL tables. When $t1$ and $t2$ each have a column named c , the result has two columns named c . In general, when $t1$ has m columns named c and $t2$ has n , $t1 \text{ CROSS JOIN } t2$ has $m+n$ columns named c .
- Following the `FROM` clause is a `WHERE` clause, denoting an invocation of the operator `WHERE`. The operands are the table resulting from the `FROM` clause and the condition following the word `WHERE`. SQL's `WHERE` operator is equivalent to **Tutorial D**'s operator of the same name when its table operand represents a relation.
- The result of the `FROM` clause has two columns of the same name, `StudentId`. The condition specified in the `WHERE` clause uses *range variables*, `IC` and `IE`, to distinguish between these two columns. The distinction is possible here, thanks to the fact that the same column name isn't used more than once in either of the two operand tables (as we shall see later, that is a condition that does not always apply, even though the same column name cannot be used more than once in a base table).
- The range variables are defined in the `FROM` clause alongside the table expressions to which they apply. The key word `AS` separating the table expression from the range variable name is optional. If the table expression consists of just a table name, unaccompanied by a range variable, then that table name serves also as a range variable name.
- A range variable is so-called because it is considered to “range over” each element in turn of a collection, the collection in the example at hand being the rows of a table. Note carefully that although the expression `IE.StudentId` is a column *reference*, it is not a column *name*. It references a particular column named `StudentId`. The prefix “`IE.`” is required because without it the column reference would be ambiguous.

Historical Notes

You may have learned a different term for *range variable*, which was used by Codd in his early papers but not adopted by the SQL standard until 2003. In some SQL texts it is called *alias* but this is not at all appropriate, really, because that would imply that it is a table name and therefore denotes a table rather than a row. The SQL standard uses the equally inappropriate term *correlation name* (it doesn't denote a correlation, whatever that might be), but only for the case where the name is explicitly given (via `AS` in the example) and not for the case where a simple table name doubles as a range variable name. In SQL:2003 range variable was adopted as a convenient single term to cover the more general case.

In his seminal 1970 paper [3] E.F. Codd defined the relational algebra, subsequently adopted in various guises by ISBL, Business System 12, and much later, **Tutorial D**. Such an algebra was clearly suited to the conventional style of most computer languages, but also in that paper Codd proposed an alternative approach based more closely on the predicate calculus. He called this notation relational calculus and in a later paper [4] proposed a concrete syntax which he called “Data Sublanguage ALPHA”.

It seems that the authors of SEQUEL took inspiration from both relational algebra and Codd’s ALPHA, as well as the style of the typical scripting languages of that time that were used for generating reports from files. (The SEQUEL paper explicitly mentions as a source of inspiration such a scripting language, the IBM product GIS—Generalized Information System.)

The idea to use range variables in SEQUEL, thence in SQL, came from ALPHA. Now, when the planners of Business System 12 (of which I was one) were studying Codd’s papers, we realized very early on that we needed to choose a style for our language. The algebra appealed because of its conventionality as a set of operators closed over something—relations, of course—and happily the prototype implementation ISBL, which used the algebra defined in reference [14], provided answers to various questions that Codd’s proposal had raised, in particular, questions concerning the headings of relations resulting from invocations of these operators.



Strømmen produseres ofte langt fra der den skal brukes.

Statnett sitt oppdrag er å gjøre strømmen tilgjengelig, uansett hvor i dette langstrakte landet du bor. Det er vi som bygger og drifter "riksveiene" i norsk strømforsyning. Gjennom vårt landsdekkende nett sørger vi for en sikker fordeling av strøm mellom nord, sør, øst og vest.

Vi binder Norge sammen

Statnett
Vårt felles kraftnett

Er du student? Les mer her
www.statnett.no/no/Jobb-og-karriere/Studenter



ALPHA also had appeal, but a similar question concerning headings arose and this time we could not find a satisfactory answer. Were these “dot qualified” names arising from the use of range variables to be the actual attribute names in the result’s heading? If so, what if that relation were input to another ALPHA expression? Did the result of that expression have doubly qualified attribute names? And so on, yielding longer and longer names with more and more dots in them? If so, well, we didn’t think that would be acceptable and in any case in those days computer memory was at a premium and for performance reasons we needed reasonably small, fixed-length storage slots to accommodate attribute names. On the other hand, if these qualifiers did not appear in the result headings, how were attributes of the same name to be distinguished? We decided against support for notation based on Codd’s calculus, because no implementation had been made to provide answers to these questions. Moreover, if and when satisfactory answers appeared, then we would have the option to provide such support as a mapping to the algebra-based language—an alternative interface for users who might prefer that style.

The authors of SEQUEL decided differently. In fact they decided to be different from both the algebra and the calculus. They decided that in a base table each column would have a unique name but that requirement would not carry through in general to results of table expressions. The presence of two or more columns with the same name didn’t matter much in SEQUEL, or in SQL prior to 1992, because table expressions other than simple table names were not permitted to appear as operands of FROM—the language was thought by some, erroneously as it turned out, to be relationally complete even with that restriction. A similar remark applies in connection with columns that have no name at all, as in `SELECT SUM(X) FROM T`, for example. If tables with such columns can arise only as results of `SELECT` expressions, and `SELECT` expressions aren’t permitted in `FROM` clauses, then there’s never any possibility to reference such columns by name, and if the language is complete, then there isn’t any need to either. The reason why relational completeness in SQL requires support for `SELECT` expressions in the `FROM` clause is given in Chapter 5, Section 5.6, **Summarization in SQL**.

Another point motivating the decisions taken in ISBL and BS12 was that it was felt that every expression denoting a relation should be assignable to a relation variable of the same type. SEQUEL was silent on the subject of variables but in SQL not every table expression can be the value of a base table, because a base table is required to have a unique name for every column.

As for the strange restrictions that govern the syntax of `SELECT` expressions, requiring a query to be expressed as a fixed sequence of invocations of specific operators, that would have been presumably inspired by GIS. The typical style of report generation languages in those days was based around something like this, assuming for simplicity that a single input file contains all the data needed for the report:

1. Specify the input file, whose physical layout in terms of fields in records would be described in something called the Data Dictionary, which gave names to those fields. So this became SEQUEL’s `FROM` clause.

2. Specify a condition to select the desired records from that file. So that became the `WHERE` clause.
3. Specify the values, derived from the selected records, that were to appear in the output report. So that became the `SELECT` clause, placed first in spite of it being actually the last step, to align the complete expression with the normal style of English sentences.

The foregoing account is largely conjecture but it seems quite reasonable from the evidence available. Recalling that the “S” in SEQUEL and SQL originally stood for “structured” (though this latter never became official), we can add the observation that “structured” was something of a buzz word in the 1970s, when the discipline of structured programming rightly became fashionable. However, there is of course no connection at all between the structure of SQL table expressions and the discipline of structured programming.

4.2 Relations and Predicates

Sections 4.2 in the theory book states the importance of *the closed world assumption* and applies just as well to SQL, insofar as it can be deemed applicable in the presence of that intrusive truth value, UNKNOWN, discussed in Chapter 3.

4.3 Relational Operators and Logical Operators

Section 4.3 in the theory book prepares the ground for subsequent sections in which each specific relational operator is paired with its logical counterpart, such that, for example, $r1 \text{ JOIN } r2$ denotes the relation representing the extension of the predicate $p1 \text{ AND } p2$, the conjunction of the predicates for the operand relations. It follows that where we can find SQL counterparts of those relational operators, invocations of those counterparts will in turn represent extensions of the predicates given in the theory book.

The definitions that appear in the following sections use the following conventions:

- Symbols beginning with t denote tables and those beginning with r denote rows. In the theory book, of course, it's the other way around, so to speak, because r stands for relation, t for tuple.
- Because the columns of a table, and therefore the components of a row, are ordered in SQL, the term *concatenation* is used in its usual sense to refer to connecting two or more things (headings or rows), one after the other in the order specified.

4.4 JOIN and AND

In the theory book Section 4.4 is all about one operator, JOIN. SQL's closest counterpart, NATURAL JOIN, has already been covered. Here we look at several other “join” operators defined in SQL. We don't really need to, as NATURAL JOIN, if considered as primitive, renders all the others redundant as shorthands. But as has already been mentioned, you won't find NATURAL JOIN in every SQL product. CROSS JOIN has already been mentioned as the operator implicitly used in joining the tables specified in a FROM clause's commalist. We start by giving its full definition.

Definition of CROSS JOIN

Let $s = t1 \text{ CROSS JOIN } t2$, where $t1$ and $t2$ are table expressions optionally accompanied by range variables. Then:

- The heading H_s of s is the concatenation of the headings of $t1$ and $t2$, in that order.
- If $r1$ is a row appearing $n1$ times in $t1$ and $r2$ is a row appearing $n2$ times in $t2$, then the row formed by concatenation of $r1$ and $r2$ in that order appears $n1 \cdot n2$ times in the body of s .

It follows that the degree of the result is the sum of the degrees of the operands and its cardinality is the product of their cardinalities, as with $r1 \text{ TIMES } r2$ in **Tutorial D**.



Hva får egentlig en ingeniør- eller teknologistudent for 300 kroner?

- Medlemskap i en aktiv studentorganisasjon – hele studietiden
- 150 tillitsvalgte studenter som ivaretar dine interesser
- Jobbsøkerkurs
- Gratis PC-forsikring og gode bank- og forsikringstilbud
- Teknisk Ukeblad og NITO Refleks
- Møteplasser på web 2.0

Flere medlemsfordeler og innmelding: www.nito.no/student

Alle som studerer på ingeniør-, bioingeniør-, sivilingeniør eller andre teknologistudier (høgskolekandidat, bachelor eller master) kan bli medlem i NITO.

NITO NORGES STØRSTE ORGANISASJON FOR INGENIØRER OG TEKNOLOGER



Interesting properties of CROSS JOIN

Compare these with the “interesting properties of JOIN” given in the theory book.

CROSS JOIN is *associative* but not *commutative*, for the reason given in Section 4.1.

Unlike JOIN and NATURAL JOIN, CROSS JOIN is not *idempotent*. Let t be any table and let n be its cardinality. Then t CROSS JOIN t has twice as many columns as t and n^2 rows.

Also unlike JOIN, CROSS JOIN has no identity value. If t is a table, there is no table $t0$ such that t CROSS JOIN $t0 = t$. That’s because $t0$ would be required to have no columns and exactly one row, but SQL doesn’t recognize the existence of tables with no columns.

We can now define FROM in terms of CROSS JOIN.

Definition of FROM

Recall that the operand of FROM is denoted by a commalist, each element of that commalist being a table expression optionally accompanied by a range variable name. Then we can write:

FROM fe_1, fe_2, \dots, fe_n ($n \geq 0$) is equivalent to FROM fe_1 CROSS JOIN fe_2 CROSS JOIN \dots CROSS JOIN fe_n .

Recall that the operands of CROSS JOIN can also include range variables. Note also that an invocation of CROSS JOIN is itself a table expression and can thus appear in an element of a FROM clause. In the light of that observation, consider Example 4.1b

Example 4.1b: FROM giving indistinguishable columns of same name

```
FROM    (IS_CALLED AS IC CROSS JOIN IS_ENROLLED_ON AS IE) AS CJ
WHERE   ?? .StudentId = ?? .StudentId
/* The StudentId columns can no longer be distinguished */
```

The only qualifier available for the columns of the result of that FROM clause is CJ. The range variables IC and IE are rendered out of scope by the definition of CJ.

Historical Notes

A restricted version of `FROM` was defined in SEQUEL and thence in the first implementations of SQL. The restriction was that each operand had to be a simple table name, referencing either a base table or a “view” (SQL’s counterpart of **Tutorial D**’s virtual relvars) rather than a table expression of arbitrary complexity. This restriction, along with others concerning the table expressions on which views were defined, is one of those that rendered SEQUEL and early SQL relationally incomplete. See Chapter 5, Section 5.6, **Summarization in SQL**.

`CROSS JOIN` was added to the international standard in 1992, along with the other explicit `JOIN` operators. It is completely redundant and can hardly be regarded as a shorthand, but perhaps its use makes certain expressions clearer than they would otherwise be. It remains an optional conformance feature.

Another variety of `JOIN` is illustrated in Example 4.1c—the “named columns join”. Here the common columns to be used for matching rows are specified explicitly in a parenthesized commalist following the word `USING`. As with `NATURAL JOIN`, each column used for matching appears just once in the result, so Example 4.1c is in fact equivalent to Examples 4.1 and 4.1a.

Example 4.1c: Obtaining a natural join by specifying the common columns

```
SELECT * FROM IS_CALLED JOIN IS_ENROLLED_ON USING ( StudentId )
```

However, a named columns join doesn’t always have an equivalent formulation using `NATURAL JOIN`. That’s because although each `USING` column must be a common column, it is not necessary to specify *all* the common columns. A common column whose name does not appear in the `USING` list gives rise to two columns of that name in the result, which therefore does not represent a relation.

To cater for cases where the columns used for matching purposes are not common columns, and/or the matching operator is not “=”, the required joining condition can be spelled out as shown in Example 4.1d, in parentheses following the key word `ON`.

Example 4.1d: Explicitly specifying the join condition

```
SELECT *  
FROM   IS_CALLED JOIN IS_ENROLLED_ON  
       ON ( IS_CALLED.StudentId = IS_ENROLLED_ON.StudentId )
```

Note carefully that Example 4.1d is *not* equivalent to Example 4.1. That’s because the result now contains two `StudentId` columns, as would of course be required if they didn’t have the same name. In fact, Example 4.1c is equivalent to the table expression obtained by replacing `JOIN` by a comma and `ON` by `WHERE`.

Now, the key word `JOIN` in all of the foregoing examples can be harmlessly preceded by the word `INNER`. SQL also supports what are called “outer joins”. The outer join of t_1 and t_2 contains all the rows of the inner join and possibly some more if either operand has rows which fail to participate in the inner join. Such a row might participate in the outer join, accompanied by `NULL` for each column of the other operand. The key words `LEFT`, `RIGHT`, and `FULL`, each optionally followed by `OUTER`, are used to specify whether unmatched rows of the first (left) operand, the second (right) operand, or both operands, respectively, are to appear in the result. Example 4.1e shows an SQL outer join. As well as the rows shown in Figure 4.2, a single row for student `S5` appears in the result, with `NULL` in place of a value for `CourseId`.

Example 4.1e: An SQL outer join

```
SELECT *
FROM   IS_CALLED NATURAL LEFT JOIN IS_ENROLLED_ON
```

Note that adding `LEFT` to an invocation of `CROSS JOIN` has no effect unless the right-hand operand table is empty. As outer joins in general denote tables that are not counterparts of relations, they merit no further discussion here.


Skatteetaten



Vil du jobbe i et av landets største IT-miljøer?

Vi skal gjøre det kompliserte enkelt

Skatteetaten tilbyr store fagmiljø og utfordrende oppgaver innen:

- > Systemutvikling
- > Service oriented architecture (SOA)
- > Business intelligence (BI)
- > Testledelse
- > Webutvikling
- > IT sikkerhet
- > Infrastruktur
- > Brukergrensesnitt

For nyutdannede IT-spesialister kan vi tilby et to-årig traineeprogram.

For mer informasjon se skatteetaten.no/jobb

Profesjonell • Nytenkende • Imøtekommende



Historical Notes

Apart from `NATURAL`, `CROSS`, and `FULL`, all the `JOIN` options described in this section became mandatory conformance features in SQL:1999.

When outer joins were added to SQL in 1992, the use of `NULL` for the noncommon columns in unmatched rows was a fairly obvious choice. The designers of Business System 12 also recognized the requirement for outer joins but could not use `NULL` for the same purpose because the decision had already been taken not to include any such construct in that language. Instead, they defined an operator, `MERGE`, similar to SQL's `LEFT JOIN` but differing from `LEFT JOIN` in two respects. First, the values to be used for the noncommon attributes of unmatched tuples had to be explicitly specified in the invocation (and they had to be values, of course—there was no such thing as `NULL`). Secondly, the common attributes were required to constitute a superkey of the second operand, thus guaranteeing that the join was many-to-one or one-to-one (i.e., each tuple of the first operand would be joined with at most one tuple of the second). In Example 4.1e the join is one-to-many, which makes the appearance of just one row for student S5 seem somewhat arbitrary. The superkey requirement did not further restrict the second operand in any way because BS12 required a key to be specified for every base relvar and, furthermore, from these declared keys, and the semantics of each relational operator, BS12 was able to infer keys for the results of invocations of relational operators.

4.5 Renaming Columns

SQL has no direct counterpart of `RENAME`. To derive the table on the right in Figure 4.4 from the table on the left, **Tutorial D** has `IS_CALLED RENAME { StudentId AS Sid }`.

StudentId	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

Sid	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

Figure 4.4: Tables differing only in a column name

Example 4.2 shows how the same effect can be achieved in SQL. Note that the `SELECT` clause has to include all the columns that are not subject to renaming, as well as those that are.

Example 4.2: Renaming a column

```
SELECT T1.StudentId AS Sid1, T1.Name, T2.StudentId AS Sid2
FROM   IS_CALLED T1, IS_CALLED T2
WHERE  T1.Name = T2.Name
      AND T1.StudentId < T2.StudentId
```

Download free eBooks at bookboon.com

Explanation 4.2

- Each element of a `SELECT` clause is an expression of arbitrary complexity that can reference one or more columns of the `SELECT` clause's input table. Note that the expression is not *required* to reference any columns: it might be just a literal, for example.
- If the expression consists of just a column reference, then the name of the referenced column is the name of the corresponding column in the result.
- If the expression is followed by the key word `AS`, then the resulting column has the column name given after that key word.

SQL allows the result to have two or more columns of the same name. For example, if we replace `Sid` by `Name` in Example 4.2, we still have a valid SQL `SELECT` expression.

Historical Notes

Column renaming with `AS` wasn't in `SEQUEL` or the early implementations of SQL. It was added to the international standard in 1992, along with the liberation of `FROM` to allow table expressions in general as operands. Without `AS`, some columns in the result of `FROM` would have to be anonymous or have nonunique names. Such columns cannot be referenced in subsequent clauses such as `WHERE` and `SELECT`.

The fact that, in standard SQL (though not in all implementations), `AS` can assign the same name to more than one column in the same `SELECT` clause may surprise you. The rationale for this is that prior to 1992 an SQL table expression could already result in a table with two or more columns of the same name (`SELECT C, C FROM T`, for example, or `SELECT T1.C, T2.C FROM T1, T2`). A prohibition on the use of `AS` for this purpose would be futile unless duplicate column names were outlawed altogether, which was out of the question for the paramount reason of backwards compatibility.

Using `RENAME` in combination with `JOIN`

Example 4.3 in the theory book gives pairs of ids of students having the same name, by joining two renamings of `IS_CALLED`. Example 4.3a gives an equivalent expression in SQL.

Example 4.3a: Renaming and joining

Student *Sid1* is called *Name* and so is student *Sid2*

```
SELECT *
FROM   (SELECT StudentId AS Sid1, Name FROM IS_CALLED)
       NATURAL JOIN
       (SELECT StudentId AS Sid2, Name FROM IS_CALLED)
```

As before, the result sagely tells us that student S1 (Anne) has the same name as herself and also shows two pairings of S1 with S5 (both named Boris). The pairing of a student id with itself can be avoided by adding `WHERE Sid1 <> Sid2` to the `WHERE` clause. The duplicate pairings can further be avoided by using `<` instead of `<>` in this addition, but that trick assumes that an ordering is defined for type `SID`, which is not necessarily the case. If `NATURAL JOIN` is not available, an expression such as the one shown in Example 4.3b could be used instead.

Example 4.3b: Renaming and joining without `NATURAL JOIN`

Student *Sid1* is called *Name* and so is student *Sid2*

```
SELECT T1.StudentId AS Sid1, T1.Name, T2.StudentID AS Sid2
FROM   IS_CALLED T1, IS_CALLED T2
WHERE  T1.Name = T2.Name
      AND T1.StudentId < T2.StudentId
```

4.6 Projection and Existential Quantification

Intuitively it might seem that projection in SQL is simply a matter of specifying the required columns in the `SELECT` clause, as in Example 4.4a.



OLJE- OG ENERGIDEPARTEMENTET



Er du full av energi?

Olje- og energidepartementets hovedoppgave er å tilrettelegge for en samordnet og helhetlig energipolitikk. Vårt overordnede mål er å sikre høy verdiskapning gjennom effektiv og miljøvennlig forvaltning av energiressursene.

Vi vet at den viktigste kilden til læring etter studiene er arbeidssituasjonen. Hos oss får du:

- Innsikt i olje- og energisektoren og dens økende betydning for norsk økonomi
- Utforme fremtidens energipolitikk
- Se det politiske systemet fra innsiden
- Høy kompetanse på et saksfelt, men også et unikt overblikk over den generelle samfunnsutviklingen
- Raskt ansvar for store og utfordrende oppgaver
- Mulighet til å arbeide med internasjonale spørsmål i en næring der Norge er en betydelig aktør

Vi rekrutterer sivil- og samfunnsøkonomer, jurister og samfunnsvitere fra universiteter og høyskoler.

www.regjeringen.no/oed



Se ledige stillinger her

www.jobb.dep.no/oed



Example 4.4a: Projection (incorrect)

Student *StudentId* is enrolled on some course.

```
SELECT StudentId FROM IS_ENROLLED_ON
```

Unfortunately, though, if some student is currently enrolled on more than one course (as indeed student S1, Anne, is in our example database), then the row giving that student's id appears twice in the result, which because of that duplicate appearance does not represent a relation. To avoid multiple appearances of the same row SQL requires you to write the word `DISTINCT`, as in Example 4.4b. (The key word `ALL` can be given instead of `DISTINCT`, clarifying that duplicate rows are not to be eliminated. As `ALL` is the default option, it is rarely seen in examples and I do not use it in this book.)

Example 4.4b: Projection (correct version)

Student *StudentId* is enrolled on some course.

```
SELECT DISTINCT StudentId FROM IS_ENROLLED_ON
```

In more complicated examples it is sometimes quite difficult to tell whether omission of `DISTINCT` would give rise to duplicate rows. It might therefore seem good advice to always write `DISTINCT`. Indeed, I would certainly advocate such practice to students having to write SQL expressions in solutions to questions in exam papers, for example, but if it were followed blindly in commercial systems, then many queries would run very much slower than need be because typical SQL implementations have little or nothing in the way of built-in intelligence to recognize cases where duplicate rows cannot possibly arise. Although such intelligence is quite feasible within acceptable limits (and was used in Business System 12, for example), the inclusion of `DISTINCT` allows SQL implementations to place the responsibility for duplicate elimination on the user.

Recall that **Tutorial D** allows projection to be expressed either by listing the attributes to be included or by listing the ones to be excluded, using `ALL BUT`. SQL has no counterpart of the `ALL BUT` variety.

Effect of NULL

Rows $r1$ and $r2$ are considered as duplicates in SQL when $r1$ IS NOT DISTINCT FROM $r2$ evaluates to `TRUE`. Recall that $r1 = r2$ can evaluate to `UNKNOWN` in such cases. So `SELECT DISTINCT` is one of those exceptional cases where, in the words of the SQL standard, “multiple null values are treated together”.

Historical Notes

The requirement for the user to specify whether duplicate rows are to be eliminated was in original SQL. In fact it was also in SEQUEL but with an interesting difference: elimination of duplicates was the default option in SEQUEL.

It might seem a simple matter to add support for column exclusion to SQL by syntax such as `SELECT * EXCEPT <column name list>`, but a proposal to that effect for the SQL standard was rejected in 2004 as a result of objections raised by the USA delegation. They argued that the use of `SELECT *` is frowned upon by most SQL experts and should be discouraged. To that end, they would accept no enhancements to `SELECT *`.

Of course, the reason why `SELECT * FROM T` is so deprecated is that its meaning changes whenever the definition of `T` is changed—for example by use of `ALTER TABLE`. Applications that always spell out exactly which columns are required in the `SELECT` clause can be to a greater extent immune to such schema changes. (A similar comment would apply to the use of **Tutorial D** expressions such as `IS_ENROLLED_ON` and `IS_ENROLLED_ON{ALL BUT CourseId}`, but the problem is addressed in **Tutorial D** by allowing an application to define its own perception of the database using a special form of local relvar, not mentioned in the theory book.) However, in SQL the columns of a table expression might be explicitly spelled out in the `FROM` clause, as in Example 4.3a, in which case the use of `SELECT *` is both convenient and harmless.

How ENROLMENT was split

Example 4.5 in the theory book shows how relvars `IS_CALLED` and `IS_ENROLLED_ON` can be derived from the original `ENROLMENT` relvar, using projection in the initial assignment to those relvars. Here is how the same effect can be achieved in SQL:

Example 4.5: Splitting ENROLMENT

```
CREATE TABLE IS_CALLED
AS (SELECT DISTINCT StudentId, Name FROM ENROLMENT)
WITH DATA ;

ALTER TABLE IS_CALLED ADD CONSTRAINT PRIMARY KEY ( StudentId ) ;

CREATE TABLE IS_ENROLLED_ON
AS (SELECT DISTINCT StudentId, CourseId FROM ENROLMENT)
WITH DATA ;

ALTER TABLE IS_ENROLLED_ON ADD CONSTRAINT PRIMARY KEY ( StudentId, CourseId ) ;

DROP TABLE ENROLMENT ;
```

Explanation 4.5:

- **CREATE TABLE IS_CALLED** announces that what follows defines a base table named `IS_CALLED`.
- **AS (SELECT DISTINCT StudentId, Name FROM ENROLMENT)** specifies that the columns of `ENROLMENT` and their declared types are as in the specified expression.
- **WITH DATA** additionally specifies that the table resulting from the specified expression is to be the initial value of `IS_CALLED`.
- **ALTER TABLE IS_CALLED ADD PRIMARY KEY (StudentId)** specifies a constraint to the effect that no two distinct rows having the same `StudentId` value can ever appear simultaneously in `IS_CALLED`. Note that this constraint has to be given as a separate statement from the one that creates the base table. If the key word `DISTINCT` had been omitted, the `CREATE TABLE` statement would have succeeded but the `ALTER TABLE` statement would have failed because the required constraint would have been violated by the two appearances of the row for student S1, Anne.
- Similar comments apply to the `CREATE` and `ALTER TABLE` statements for `IS_ENROLLED_ON`, but in the equivalent example in the theory book I noted that the specification `KEY {StudentId, CourseId}`, required by **Tutorial D**, is theoretically redundant because the entire heading is always a superkey. Here, the corresponding `ALTER TABLE` statement is not redundant because in the absence of any key constraints SQL allows the same row to appear several times simultaneously in the same base table.
- **DROP TABLE ENROLMENT** destroys the variable we have no further use for.

Two special cases of projection

In the theory book this section describes the identity projection, $r \{ \text{ALL BUT } \}$, and the projection on no attributes, $r \{ \}$, which yields `TABLE_DUM` when r is empty, otherwise `TABLE_DEE`. As we have already seen, the identity projection is represented in SQL by `SELECT * FROM t`, but SQL does not recognize the existence of columnless tables and so `SELECT FROM t` is not supported.

Historical Notes

The history surrounding `CREATE TABLE ... AS ...` is given in Chapter 2, in the Historical Notes for Section 2.11.

Various attempts have been made to develop “natural language query” products, at least in prototype form, whereby queries expressed in a human language are translated into SQL. In the 1980s I became peripherally involved in one such product. Its developers told me that they could handle questions such as “Did any student score more than 90 in the relational database theory exam?” but the translation to SQL was awkward because they had to make a special case for the generation of the `SELECT` clause, which would otherwise include no columns! The **Tutorial D** expression for such a query would end with a projection on no attributes, yielding either `TABLE_DEE` (meaning “yes”) or `TABLE_DUM` (“no”).

4.7 Restriction and AND

Restriction in **Tutorial D** is available via the `WHERE` operator, and so it is in SQL—we have already seen it several times in this chapter, such as Example 4.3b. However, the subject is introduced in the theory book by Example 4.6, showing how a certain simple restriction can be expressed using `JOIN` and a relation literal. It is useful to show SQL’s counterpart of that example, giving the student ids of students named Boris.



HELT GRATIS!

S for Skikk & Bank

En bok om ting som er greit å vite når du har flyttet hjemmefra.

dnb.no

DNB

Bank fra A til Å



Example 4.6: joining with a table literal

```
SELECT DISTINCT StudentId
FROM    IS_CALLED NATURAL JOIN ( VALUES ( 'Boris' ) ) AS T(Name)
```

As noted in Chapter 2, the columns of a table literal in SQL are anonymous. When a table literal is an operand in a table expression the only way of assigning names to its columns is by giving them in the definition of a range variable—AS T(Name) in the example. Example 4.7 shows the equivalent, more intuitive formulation using WHERE.

Example 4.7: Restriction in SQL

```
SELECT DISTINCT StudentId
FROM    IS_CALLED
WHERE   Name = 'Boris'
```

The WHERE clause operates on the result of the FROM clause in analogous fashion to the way it operates on an arbitrary relation expression in **Tutorial D**. Name = NAME ('Boris') is an open expression, as defined in the theory book. The key word DISTINCT here is redundant, as it happens, because StudentId is a declared key for IS_CALLED.

Example 4.8 in the theory book finds students whose names start with B. Example 4.8 here shows the SQL counterpart, this being one that cannot feasibly be expressed by joining with a table literal.

Example 4.8: A more useful restriction

```
SELECT *
FROM    IS_CALLED
WHERE   SUBSTRING(Name FROM 1 FOR 1) = 'B'
```

Note in passing the unconventional syntax used to invoke the built-in SUBSTRING operator. An alternative way of expression the WHERE condition, and one that has been in SQL for longer than SUBSTRING, which first appeared in SQL:1992, is Name LIKE 'B%', in which the % character is used as a “wild card” signifying “anything can appear here, even nothing at all”—nowadays, in search engines and the like, * is more commonly used for such purposes.

Effects of NULL

A note of caution needs to be made here. Recall that in general a conditional expression in SQL can evaluate to UNKNOWN as well as TRUE or FALSE. A row satisfies the WHERE condition c if and only if c is TRUE. This is inconsistent with SQL's treatment of database constraints, which are deemed to be satisfied whenever they evaluate to either TRUE or UNKNOWN. Consider, for example, a constraint declared with condition `NOT EXISTS (SELECT * FROM T WHERE X <= Y)`, requiring every row in T to satisfy the condition $X > Y$. Then it can happen that `SELECT * FROM T WHERE X > Y` is empty—for example, if T is not empty but $X \text{ IS NULL}$ is true for every row—even though `SELECT * FROM T` is not empty!

Historical Note

The `SUBSTRING` operator was added to SQL in 1992 and became a mandatory conformance feature in SQL:1999. The motivation behind the unconventional syntax for invoking `SUBSTRING` and other built-in operators lay in a desire to distinguish invocations of built-in operators syntactically from those of user-defined operators.

4.8 Extension and AND

The theory book gives the following simple example of relational extension in **Tutorial D**:

```
EXTEND IS_CALLED : { Initial := FirstLetter ( Name ) }
```

Assuming the user-defined operator `FirstLetter` is available to the SQL user, this can be expressed easily in SQL but there is a strange quirk in the grammar at play here:

```
SELECT IC.*, FirstLetter ( Name ) AS Initial
FROM    IS_CALLED AS IC
```

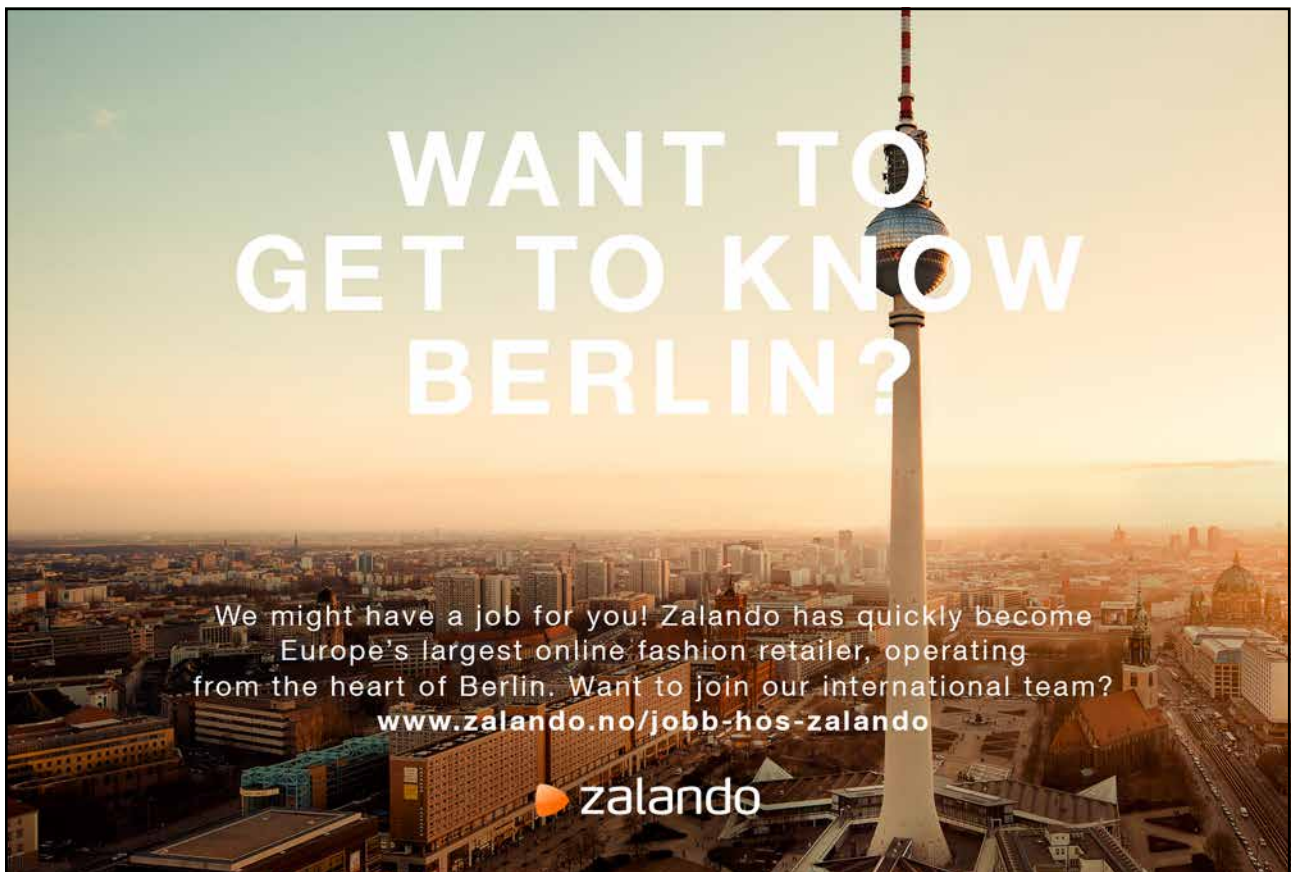
Note very carefully that we have to qualify the `*` using the range variable, `IC`, which in this case ranges over the rows of the current value of `IS_CALLED`. When we use a `SELECT` clause to “add columns” to the table on which it operates, it seems obvious to write `*` to specify that every column of that operand table is required and to follow it with a commalist of expressions denoting the additional columns. For some reason the official SQL grammar does not allow additional columns to accompany an unadorned `*`. When the `FROM` clause contains several entries, pure extension becomes more difficult to express in SQL. For example, if the `FROM` clause defines range variables `T1`, `T2`, and `T3`, we could write `SELECT T1.*, T2.*, T3.* ...`, but that would defeat the purpose. To be able to write just a single `*` to denote all the columns of the `FROM` table, we would have to resort to something like

```
SELECT T.*, ...
FROM    ( SELECT * FROM T1, T2, T3 WHERE ... ) AS T
```

Quirks like this in the SQL grammar serve only to further exacerbate the difficulties for teachers and students caused by the language's major departures from the theory that I have already described.

Effects of **NULL**

In general, if some argument to a read-only operator invocation is **NULL**, then so is the result of that invocation. For example, $X + Y$ evaluates to **NULL** if either $X \text{ IS NULL}$ or $Y \text{ IS NULL}$ evaluates to **TRUE**. There are some exceptions, IS NULL being an obvious one. User-defined operators, of course, can make their own arrangements—my putative `FirstLetter` operator, for example, might be defined to return the empty string, `' '`, when its argument is **NULL**.



Historical Notes

Codd did not include an extension operator in his algebra. The reason he gave for this omission did not stand up to scrutiny. He regarded the ability to compute such “additional values” as the responsibility of the application program, using facilities of the host language rather than his proposed “data sublanguage”. Database practitioners knew, of course, that this approach was hopelessly flawed and the designers of ISBL and SEQUEL both came up with fairly obvious solutions to rectify matters. ISBL being based firmly on explicit relational operators, it was a simple matter to invent extension. It was no doubt equally easy for SEQUEL to allow “additional columns” to be specified in the `SELECT` clause, but their decision not to require or even allow the user to provide names for those columns should surely have been brought into question by the designers of SQL.

As for the quirk concerning the use of `*`, a proposal to allow an unadorned `*` to be accompanied by additional entries in the `SELECT` clause was submitted in 2004 (along with the previously mentioned proposal to support `* EXCEPT`) but was rejected in the face of the same objection (use of `*` is deprecated and should not be encouraged by the introduction of further enhancements).

4.9 UNION and OR

SQL supports `UNION` explicitly but differently from the way it supports `JOIN` explicitly. As we have seen, `JOIN` is used exclusively within the `FROM` clause, such that `IS_CALLED NATURAL JOIN IS_ENROLLED_ON`, for example, can be an element of that clause but cannot stand alone as a table expression. Instead, `UNION` always connects table expressions that *can* stand alone, these being:

- `SELECT` expressions
- `TABLE tn`, which is equivalent to `SELECT * FROM tn`, where *tn* is a table name
- `VALUES` expressions
- Invocations of `UNION`, `INTERSECT` (see Chapter 5, Section 5.2, **Semijoin and Composition**) and `EXCEPT` (see section 4.10, **Semidifference and NOT**)

Actually, just as SQL has several varieties of `JOIN`, it also has several varieties of `UNION`, none of which is equivalent to the relational operator of that name. The closest approximation to relational union is illustrated in Example 4.9a, a translation to SQL of the theory book’s Example 4.9.

Example 4.9a: SQL's closest approximation to relational union

```
SELECT StudentId
FROM    IS_CALLED
WHERE   Name = 'Devinder'
UNION DISTINCT CORRESPONDING
SELECT StudentId
FROM    IS_ENROLLED_ON
WHERE   CourseId = 'C1'
```

The key word `DISTINCT` is optional and implied by default (somewhat curiously so, considering that its opposite, `ALL`, is the default option in the `SELECT` clause). It specifies that no row is to appear more than once in the result. Thus, there is never a need to include `DISTINCT` in either of the `SELECT` clauses, and this would be the case even if the `WHERE` clause were omitted from the specification of the second operand in Example 4.9a, allowing the same `StudentId` value to appear more than once in that operand.

The key word `CORRESPONDING` specifies that operand columns are to be paired by name, just as in relational union. Thus, the slightly revised version shown in Example 4.9b is also legal and is in fact equivalent to Example 4.9a. Curiously, the corresponding columns do not have to be of the same type! However, each value appearing in a corresponding column of the second operand must be “castable” to a value in the type of its counterpart in the first operand. For example, the character string “+123” is castable to the value 123 of type `INTEGER` but the character string “123.5” is not.

Example 4.9b: Union applied to disparate operands

```
SELECT *
FROM    IS_CALLED
WHERE   Name = 'Devinder'
UNION DISTINCT CORRESPONDING
SELECT *
FROM    IS_ENROLLED_ON
WHERE   CourseId = 'C1'
```

This is legal and equivalent because `CORRESPONDING` specifies that *only* the common columns of each operand are to appear in the result. In this case `StudentId` is the only common column, of course. As usual, the common columns of one operand must be of the same type as their counterparts in the other operand. And as you might expect by now, there has to be at least one common column (SQL doesn't recognize the existence of columnless tables).

A further variation, also equivalent to Example 4.9a, is shown in Example 4.9c.

Example 4.9c: Specifying the columns required

```
SELECT *  
FROM   IS_CALLED  
WHERE  Name = 'Devinder'  
UNION DISTINCT CORRESPONDING BY (StudentId)  
SELECT *  
FROM   IS_ENROLLED_ON  
WHERE  CourseId = 'C1'
```

The required columns can be specified explicitly in a parenthesized commalist following the key word `BY`. Columns thus specified must be common to both operands but the list is not required to specify *all* the common columns.

I turn now to the use of `UNION` without `CORRESPONDING`. Example 4.9d is derived from 4.9a merely by omitting `CORRESPONDING` and is in fact equivalent to 4.9a, but only because the operands have identical `SELECT` clauses.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"

Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



Example 4.9d: UNION without CORRESPONDING

```
SELECT StudentId
FROM   IS_CALLED
WHERE  Name = 'Devinder'
UNION DISTINCT
SELECT StudentId
FROM   IS_ENROLLED_ON
WHERE  CourseId = 'C1'
```

When `CORRESPONDING` is omitted, names are not used at all in the pairing of columns. Instead, SQL's definition, in yet another departure from relational database theory, depends on an ordering of the columns: the first column of the first operand is paired with the first column of the second operand, the second with the second, and so on. As with `CORRESPONDING`, columns thus paired do not have to be of the same type. Furthermore, the two operand tables must have the same number of columns, so that there is no unpaired column in either operand, also as in relational union.

Although the operand columns in 4.9d still have the same name, `StudentId`, that is not a requirement in this variety of `UNION`. For example, `SELECT StudentId AS X` could be the `SELECT` clause of the second operand. However, if corresponding columns do not have the same name, then the corresponding column in the result is effectively anonymous (the standard defines it to have an unpredictable system-generated name). Actually, some implementations use the column names of the first operand here, thus destroying the normal commutativity of `UNION`. The user of an implementation that strictly follows the standard would perhaps be well advised always to make sure the corresponding columns have the same name anyway, to avoid the unpredictability of system-generated names and to improve portability from one implementation to another.

Further varieties of `UNION` arise when we replace the key word `DISTINCT` by `ALL` in any of the foregoing examples, as in Example 4.9e. `ALL` specifies that if row r appears n times in one operand and m times in the other, then it appears $n+m$ times in the result—*i.e.*, no elimination of duplicate rows takes place.

Example 4.9e: UNION ALL

```
SELECT StudentId
FROM   IS_CALLED
WHERE  Name = 'Devinder'
UNION ALL
SELECT StudentId
FROM   IS_ENROLLED_ON
WHERE  CourseId = 'C1'
```

Clearly, `UNION ALL` represents another departure from relational theory. However, it is commonly used when the operands can be guaranteed to be disjoint because in such cases omission of `ALL` would incur the possibly significant overhead of the duplicate elimination process with no effect on the final result.

Some authorities have argued that there really ought to be yet another variety of `UNION`, such that if row r appears n times in one operand and m times in the other, with $m \geq n$, then it appears m times in the result. Relational devotees might smile at this observation but refrain from comment.

Effect of `NULL`

In the case of `t1 UNION DISTINCT t2`, row r appears in the result, just once, if and only if either $t1$ or $t2$ contains at least one appearance of row s such that `r IS NOT DISTINCT FROM s` evaluates to `TRUE`. In other words, `NULL` is treated as equal to itself for the purposes of duplicate elimination.

Historical Notes

The grammar given in the `SEQUEL` paper uses the mathematical symbol \cup for union and defines just this single version. The body of the paper gives only a brief mention of this operator, apparently implying that its usual mathematical definition is intended.

Original SQL included just `UNION` and `UNION ALL`. `CORRESPONDING` was added in 1992 but remains an optional feature. Support for using the key word `DISTINCT` instead of just omitting `ALL` was added in SQL:1999 but remains an optional feature.

4.10 Semidifference and `NOT`


In this section in the theory book I first describe the relational difference operator, named `MINUS` in **Tutorial D**. Example 4.10 here shows SQL's closest counterpart of that operator.

Example 4.10: Difference in SQL

```
SELECT StudentId
FROM   IS_CALLED
WHERE  Name = 'Devinder'

EXCEPT DISTINCT CORRESPONDING
SELECT StudentId
FROM   IS_ENROLLED_ON
WHERE  CourseId = 'C1'
```

The syntax for **EXCEPT** exactly parallels that for **UNION**. The key words **DISTINCT**, **ALL**, and **CORRESPONDING** have exactly the same significance as in **UNION**, and **DISTINCT** remains the default option. When **CORRESPONDING** is not given, columns are paired by ordinal position, as in **UNION**.



WHILE YOU WERE SLEEPING...

www.fuqua.duke.edu/whileyouweresleeping

DUKE
THE FUQUA
SCHOOL
OF BUSINESS



$t1 \text{ EXCEPT DISTINCT } t2$ returns the table consisting of a single appearance of each row that appears in $t1$ but not in $t2$. $t1 \text{ EXCEPT ALL } t2$ returns the table consisting of $n-m$ appearances of each row that appears n times in $t1$ and m times in $t2$, with $n > m \geq 0$.

Thanks to the implicit exclusion of noncommon attributes, `EXCEPT CORRESPONDING` can also sometimes be used to obtain a semidifference ($r1 \text{ NOT MATCHING } r2$ in **Tutorial D**), but only when every column of the first operand is a common column. That is not the case in the theory book's Example 4.11, `IS_CALLED NOT MATCHING IS_ENROLLED_ON`. In general, therefore, semidifference needs to be expressed in a more elaborate longhand in SQL. Example 4.11a does it by joining the result of Example 4.10 with `IS_CALLED`. Example 4.11b does it by using SQL's comparison operator `NOT IN`, meaning “is not a member of”, in a `WHERE` condition. Example 4.11c

Example 4.11a: Semidifference via `EXCEPT` and `JOIN`

```
SELECT *
FROM   (SELECT StudentId
        FROM    IS_CALLED
        WHERE   Name = 'Devinder'
        EXCEPT DISTINCT CORRESPONDING
        SELECT StudentId
        FROM    IS_ENROLLED_ON
        WHERE   CourseId = 'C1') AS T1
        NATURAL JOIN IS_CALLED
```

Example 4.11b: Semidifference via `NOT IN` and a subquery

```
SELECT StudentId
FROM   IS_CALLED
WHERE  Name = 'Devinder'
      AND StudentId NOT IN (SELECT StudentId
                            FROM    IS_ENROLLED_ON
                            WHERE   CourseId = 'C1')
```

Example 4.11c: Semidifference via “quantified comparison” and a subquery

```
SELECT StudentId
FROM   IS_CALLED
WHERE  Name = 'Devinder'
      AND StudentId <> ALL (SELECT StudentId
                            FROM    IS_ENROLLED_ON
                            WHERE   CourseId = 'C1')
```

The `NOT IN` expression in Example 4.11b appears to be testing for the appearance of a character string in a table, but in fact the first operand *in this context* is short for `ROW (StudentId)`—and recall from Chapter 2 that the key word `ROW` is optional, even when the row to be specified has more than one column value. Thus the expression tests for the nonappearance of a given row in a given table. (You won't be surprised to hear that if the word `NOT` is omitted, then the expression becomes a test for appearance rather than nonappearance.)

In Example 4.11c `<> ALL` replaces `NOT IN` and in the absence of `NULL` has the same effect. It reads, somewhat ambiguously, as “not equal to all”. In fact the expression yields `TRUE` if and only if the condition comparing `StudentId` values is `TRUE` for every row in the result of the subquery.

Effects of NULL

The treatment of `NULL` in invocations of `EXCEPT` is as for `UNION`. This is different from its treatment in those of `NOT IN` and quantified comparisons. In the case of `t1 EXCEPT t2`, row r of $t1$ appears in the result if and only if there does not exist a row s in $t2$ such that `r IS NOT DISTINCT FROM s` evaluates to `TRUE`. Note that `x NOT IN (t)` evaluates to `UNKNOWN` whenever `x = x` does, including in particular the case where `x IS NULL` evaluates to `TRUE` because every field of the row x is the null value. Recall that when the condition given in a `WHERE` clause evaluates to `UNKNOWN` for row r , then r does not appear in the result.

Now, suppose that `IS_ENROLLED_ON` contains the rows `('S4', 'C1')`, `(NULL, 'C1')`, and no other rows for course `C1`. Then `'S4' NOT IN (SELECT StudentId FROM IS_ENROLLED_ON WHERE CourseId = 'C1')` evaluates to `FALSE`, but `'S4' <> ALL (SELECT StudentId FROM IS_ENROLLED_ON WHERE CourseId = 'C1')` evaluates to `UNKNOWN`. So examples 4.11b and 4.11c are not equivalent in the presence of `NULL`. Nevertheless, Examples 4.11b and 4.11c are equivalent because `WHERE` treats `FALSE` and `UNKNOWN` alike—a `WHERE` condition applied to a row has to yield `TRUE` for the row to appear in the result.

Historical Notes

The grammar given in the appendix to the SEQUEL paper uses the mathematical symbol “-” as an alternative to \cup for union, strongly suggesting that it stands for set difference. There is no mention of this operator in the body of the paper. However, `EXCEPT` was missing from original SQL and didn't appear in the standard until 1992. Curiously, `EXCEPT` without `ALL` is now a mandatory conformance feature while `EXCEPT ALL` and both varieties of `INTERSECT` (`ALL` and `DISTINCT`) remain optional ones. (`INTERSECT` is mentioned in the discussion on semijoin in Chapter 5, Section 5.2.) The membership tests using `IN` and `NOT IN` were in original SQL but not in SEQUEL. SEQUEL did, however, support quantified comparisons, those these were limited to rows and tables of degree one.

4.11 Concluding Remarks

I have described how the following relational operators are supported, directly or indirectly, in SQL, noting various quirks in the language on the way.

- JOIN (via `NATURAL JOIN`)
- RENAME (via possibly laborious longhand)
- projection (via `SELECT DISTINCT`)
- restriction (`WHERE`)
- EXTEND (via possibly laborious longhand)
- UNION (via `UNION DISTINCT CORRESPONDING`)
- semidifference (`NOT MATCHING` in **Tutorial D**—via `EXCEPT` for special cases, otherwise via possibly laborious longhand)



Vi vokser i Norge
og har virksomhet
helt frem til 2050

Er du interessert i sommerjobb
eller fast stilling?

Se informasjon om sommerjobber på
www.bp.no



The fact that SQL does support all of these operators, one way or another, makes SQL relationally complete, apart from its failure to support `TABLE_DEE` and `TABLE_DUM`, but I have noted several present-day features that were not in the early versions of the language, claiming that as a consequence those early versions were, unintentionally, not relationally complete. I have not yet fully explained that claim. That's because my explanation involves some of the material of the next chapter, where I look for SQL counterparts of those extra relational operators we include as “shorthands” in **Tutorial D**. See Section 5.6, **Summarization in SQL**.

EXERCISES

1. Figure 4.13 shows the supplier-and-parts database from Chris Date's *Introduction to Database Systems* (8th edition), as shown on the inside back cover of that book (except that the attribute names there are in upper case). The exercises assume you have access to an SQL implementation.

S

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

S#	P#	Qty
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

P#	Pname	Color	Weight	City
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

Figure 4.13: The suppliers-and-parts database

Execute an SQL `CREATE TABLE` statement for each of `S`, `P` and `SP`. Use `INTEGER` as the declared type for `STATUS` and `QTY`, `DECIMAL(5, 2)` for `WEIGHT`, and appropriate `VARCHAR` or `CHAR` types for all the other columns. Feel free to use lower case or mixed case to suit your own taste for column and table names, but do not otherwise change any of the given names.

Include primary key constraints as indicated by the underlining of column names in Figure 4.13.

“Populate” (as they say) each table with the values shown in Date's tables, using SQL `INSERT` statements.

2. Attempt to insert a row into `SP` with supplier number `S1`, part number `P1` and quantity `100`. Explain the result of your attempt.

3. For each of the following **Tutorial D** expressions (taken from Exercise 5 in the theory book's exercises headed **Working with a Database in Rel**), give an SQL expression that's as nearly equivalent as possible.

- a) `SP WHERE P# = 'P2'`
- b) `S { ALL BUT Status }`
- c) `SP { S#, Qty }`
- d) `P NOT MATCHING (SP WHERE S# = 'S2')`
- e) `S MATCHING (SP WHERE P# = 'P2')`
- f) `S { City } UNION P { City }`
- g) `S { City } MINUS P { City }`
- h) `((S RENAME { City AS SC }) { SC }) JOIN
((P RENAME { City AS PC }) { PC })`

4. Write SQL expressions for the following queries. Compare your solutions with its counterpart in your solutions to Exercise 6 in the theory book's exercises headed **Working with a Database in Rel**.

- a) Get all shipments.
- b) Get supplier numbers for suppliers who supply part P1.
- c) Get suppliers with status in the range 15 to 25 inclusive.
- d) Get part numbers for parts supplied by a supplier in Paris.
- e) Get part numbers for parts not supplied by any supplier in Paris.
- f) Get city names for cities in which at least two suppliers are located.
- g) Get all pairs of part numbers such that some supplier supplies both of the indicated parts.
- h) Get supplier numbers for suppliers with a status lower than that of supplier S1.
- i) Get supplier-number/part-number pairs such that the indicated supplier does not supply the indicated part.

5 Building on The Foundation

5.1 Introduction

Having described, in Chapter 4, operators that are both necessary and sufficient for relational completeness, the theory book builds on that foundation in Chapter 5 by describing some additional operators that have been found useful and are included in **Tutorial D**. These include some additional relational operators that are presented as *shorthands*—operators that can be defined in terms of ones already defined. For example, in Section 5.1, composition and semijoin are both defined in terms of the primitive operators, join and projection.

Here I show SQL counterparts for the **Tutorial D** expressions in Chapter 5, sometimes using longhands perforce. As we shall see, quite a few of the SQL examples use operators that didn't exist in SQL until 1999 or later. Moreover, these late additions are not shorthands—before they arrived, some of the examples in Chapter 5 of the theory book could not be expressed at all in SQL.

In the theory book at this point we needed a couple of additional relvars in our example database, `COURSE` and `EXAM_MARK`, along with the existing `IS_CALLED` and `IS_ENROLLED_ON`. They are shown again here, now as SQL tables, in Figure 5.1.



COURSE		EXAM_MARK		
CourseId	Title	StudentId	CourseId	Mark
C1	Database	S1	C1	85
C2	HCI	S1	C2	49
C3	Op systems	S1	C3	85
C4	Programming	S2	C1	49
		S3	C3	66
		S4	C1	93

Figure 5.1: Current values of base tables COURSE and EXAM_MARK

The predicate for COURSE is “Course *CourseId* is entitled *Title*.” The predicate for EXAM_MARK is “Student *StudentId* sat the exam for course *CourseId* and scored *Mark* marks for that exam.” The SQL definitions for these tables are

```
CREATE TABLE COURSE ( CourseId CID,
                        Title VARCHAR(100) NOT NULL,
                        PRIMARY KEY ( CourseId )
                      ) ;

CREATE TABLE EXAM_MARK ( StudentId SID,
                           CourseId CID,
                           Mark INTEGER NOT NULL,
                           PRIMARY KEY ( StudentId, CourseId )
                        ) ;
```

(SID and CID are now assumed to be SQL domain names, defined on type VARCHAR(5). As not many SQL implementations support domains, in practice you would be more likely to see some CHAR type being used here. Also, the table definitions might include some additional constraint definitions, but those are dealt with in Chapter 6.)

5.2 Semijoin and Composition

For semijoin **Tutorial D** has the dyadic relational operator **MATCHING**, defined thus:

$r1$ **MATCHING** $r2$, where $r1$ and $r2$ are relations such that $r1 \text{ JOIN } r2$ is defined, is equivalent to
 $(r1 \text{ JOIN } r2) \{ r1\text{-attrs} \}$
 where $r1\text{-attrs}$ is a commalist containing all and only the attribute names of $r1$.

and the example

```
COURSE MATCHING EXAM_MARK
```

is given as a relational expression for the predicate, “There exist a student *StudentId* and a mark *Mark* such that *StudentId* sat the exam and scored *Mark* marks for course *CourseId* and *CourseId* is entitled *Title*” (which could be abbreviated to “At least one student sat the exam for Course *CourseId*, entitled *Title*”). The resulting relation consists of just those tuples in `COURSE` that have at least one matching tuple in `EXAM_MARK`.

In SQL semijoins usually have to be done in longhand and there are several ways of doing them. The simplest way is to use the comparison operator `IN`, SQL’s counterpart of **Tutorial D**’s \in (which *Rel* implements as `IN`):

```
SELECT * FROM COURSE
WHERE CourseId IN ( SELECT CourseId From EXAM_MARK )
```

or, equivalently, the quantified comparison operator, `=ANY`:

```
SELECT * FROM COURSE
WHERE CourseId =ANY ( SELECT CourseId From EXAM_MARK )
```

The result in either case is the table shown in Figure 5.2. Note in passing that the table operand of `IN` or `= ANY` here contains three appearances of the row for course `C1`, and thus does not represent a relation. That could be fixed by specifying `DISTINCT`, of course, but here the redundant duplicates are obviously not a problem. Note the need to write the column name `CourseId` twice. Perhaps this burden is compensated for to some extent by the improved clarity.

CourseId	Title
C1	Database
C2	HCI
C3	Op systems

Figure 5.2: Semijoin of `COURSE` with `EXAM_MARK`

Translating the SQL longhand back into **Tutorial D**, we get

```
COURSE WHERE TUPLE{CourseId CourseId}  $\in$  EXAM_MARK {CourseId}
```

In spite of appearances, `TUPLE{CourseId CourseId}` is a reasonable translation of SQL's plain `CourseId` because in standard SQL the first operand of `IN` is in fact a row. A row is denoted in general by a commalist of expressions enclosed in parentheses, optionally preceded by the key word `ROW`, but the parentheses can be omitted when the row consists of just one field, as explained in Chapter 2, Section 2.9 **Table Literals**. However, many SQL implementations fail to support rows with more than one field as the first operand of `IN`, in which case a longer longhand is needed. For example, to obtain enrolments for which an exam mark is available (`IS_ENROLLED_ON MATCHING EXAM_MARK` in **Tutorial D**), we might expect to be able to write

```
SELECT * FROM IS_ENROLLED_ON
WHERE (CourseId, StudentId) IN
      ( SELECT CourseId, StudentId From EXAM_MARK )
```

but instead we are forced to write, in such an implementation,

```
SELECT * FROM IS_ENROLLED_ON A
WHERE EXISTS ( SELECT * FROM EXAM_MARK B
              WHERE A.CourseId = B.CourseId
              AND A.StudentId = B.StudentId )
```

gaiteye
Challenge the way we run

EXPERIENCE THE POWER OF
FULL ENGAGEMENT...

.....

RUN FASTER.
RUN LONGER..
RUN EASIER...

READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM



Also, in the formulation using `IN`, note carefully the need to pay attention to order in which column names are written—the correspondence between the fields of the row and the columns of the table is by position, not name, and in any case the fields of the row do not necessarily inherit the names `CourseId` and `StudentId`, as we shall see in many more examples in this chapter.

Alternatively, by translating the definition given for `semijoin` into SQL, we could write

```
SELECT DISTINCT A.*
FROM   IS_ENROLLED_ON AS A NATURAL JOIN EXAM_MARK AS B
```

and now the need to mention column names has gone away.

Finally, in the special case where all the columns of the first operand are common columns, then we can use SQL's `INTERSECT` operator. As this is indeed the case with the example at hand, we can write

```
SELECT * FROM IS_ENROLLED_ON
INTERSECT CORRESPONDING
SELECT StudentId, CourseId FROM EXAM_MARK
```

The syntax for `INTERSECT` exactly parallels that for `UNION` and `EXCEPT`. The key words `DISTINCT`, `ALL`, and `CORRESPONDING` have exactly the same significance as in those operators, and `DISTINCT` remains the default option. When `CORRESPONDING` is not given, columns are paired by ordinal position, as in `UNION`.

$t1 \text{ INTERSECT } \text{DISTINCT } t2$ returns the table consisting of a single appearance of each row that appears in both $t1$ and $t2$. $t1 \text{ INTERSECT } \text{ALL } t2$ returns the table consisting of m appearances of each row r that appears in both $t1$ and $t2$, where m is the smaller of its number of appearances in $t1$ and its number of appearances in $t2$.

Historical Notes

The SEQUEL paper uses the mathematical symbol “ \cap ” in place of SQL's `INTERSECT`. However, like `EXCEPT`, `INTERSECT` was missing from original SQL and didn't appear in the standard until 1992. It remains an optional conformance feature. The `IN` operator was in original SQL but had no counterpart in SEQUEL. Standard SQL allows `(CourseId, StudentId) = ANY` in place of `(CourseId, StudentId) IN` and defines the two formulations to be equivalent. SEQUEL's support for such “quantified comparisons” was limited to rows and tables of degree one. Also, SEQUEL did not use a key word such as `ANY` (standard SQL admits `ANY` or `SOME`, synonymously) but rather assumed existential quantification when the second operand of a comparison operator was a table and `ALL` was not specified.

Turning now to the operator known as composition, the theory book gives the example

```
COURSE COMPOSE EXAM_MARK
```

as representing the predicate “Student *StudentId* scored *Mark* marks in the exam for a course entitled *Title*.” The corresponding relation must have attributes *StudentId*, *Mark*, and *Title*. The first two would clearly be derived from *EXAM_MARK*, the third from *COURSE*.

The result is shown in Figure 5.3. As you can see, it is equivalent to the join of *COURSE* and *EXAM_MARK*, projected over all but the common attribute, *CourseId*.

Title	StudentId	Mark
Database	S1	85
HCI	S1	49
Op systems	S1	85
Database	S2	49
Op Systems	S3	66
Database	S4	93

Figure 5.3: Composition of *COURSE* and *EXAM_MARK*

In SQL, as the result contains data from both operand tables, this time we really do have to specify both tables in the *FROM* clause—neither of the operators *IN* and *EXISTS* is of any use here.

```
SELECT DISTINCT Title, StudentId, Mark
FROM    IS_ENROLLED_ON NATURAL JOIN EXAM_MARK
```

Now, it is worth repeating here the theory book’s justification for including *COMPOSE* in **Tutorial D**.

In case you are wondering if *COMPOSE* really is useful enough to be worth including in a computer language, and therefore to be worthy of inclusion in textbooks like this one, an important part of the motivation for its inclusion in **Tutorial D** was a desire to illustrate the *extensibility* of a well-designed language. Adding new operators increases a language’s complexity, to be sure, but that added complexity can be compensated for if the new operators are not only useful but can be easily defined and taught in terms of what the user already knows.

We can note in passing that it would be easy to extend SQL to support semijoin and composition explicitly—just allow words such as *MATCHING* and *COMPOSE* to appear where *NATURAL JOIN* is currently permitted!

Effects of NULL

Observations similar to those given in connection with UNION and EXCEPT in Chapter 4, Sections 4.9 and 4.10, apply here too. For example, consider the following two expressions:

```
SELECT x FROM T1
INTERSECT
SELECT x FROM T2

SELECT x FROM T1
WHERE x IN
( SELECT x FROM T2 )
```

If row r appears in both $T1$ and $T2$ and satisfies the condition $x \text{ IS NULL}$, then r appears in the result in the first case, using INTERSECT, but not in the second, because the condition in the WHERE clause evaluates to UNKNOWN for that row.



Strømmen produseres ofte langt fra der den skal brukes.

Statnett sitt oppdrag er å gjøre strømmen tilgjengelig, uansett hvor i dette langstrakte landet du bor. Det er vi som bygger og drifter "riksveiene" i norsk strømforsyning. Gjennom vårt landsdekkende nett sørger vi for en sikker fordeling av strøm mellom nord, sør, øst og vest.

Vi binder Norge sammen

Statnett
Vårt felles kraftnett

Er du student? Les mer her
www.statnett.no/no/Jobb-og-karriere/Studenter



5.3 Aggregate Operators

SQL supports all of the aggregate operators mentioned in the theory book and many more besides. The syntax, however, involves an unusual trick that SQL calls a *scalar subquery*. A scalar subquery is a table expression that satisfies all of the following conditions:

- It is enclosed in parentheses.
- It appears where a scalar expression is expected.
- The result of the enclosed table expression has exactly one column and at most one row.

The result of a scalar subquery is then either the single column value appearing in that single row or, when there is no row, `NULL`. Examples 5.1 to 5.4 are the SQL counterparts of those examples in the theory book. In each case the required aggregation is specified in the `SELECT` clause of a scalar subquery and the table operand is specified in the clause(s) following that `SELECT` clause.

Example 5.1: Counting the rows in `EXAM_MARK`

```
( SELECT COUNT(*) FROM EXAM_MARK )
```

The table expression inside the parentheses in Example 5.1 is analogous to an invocation of `SUMMARIZE` in **Tutorial D** that specifies `BY { }`, as in

```
SUMMARIZE EXAM_MARK BY { } : { X := COUNT( ) }
```

and in fact standard SQL allows `GROUP BY ()` to be added immediately before the closing parenthesis:

```
( SELECT COUNT(*) FROM EXAM_MARK GROUP BY ( ) )
```

I could have added `AS X` to the `SELECT` clause, to make the analogy exact, but that would have been pointless because of course the column name disappears in the process of converting the table to a number. Similar comments apply to Examples 5.2, 5.3, and 5.4. (SQL's `GROUP BY` construct is described in Section 5.6.)

Remember that the SQL Example 5.1 denotes a number, as opposed to a table, only when the context is appropriate—for example, when it appears on the right-hand side of an assignment to an integer variable, or as an operand in a comparison of two integers. When it appears as an element of a `FROM` clause, for example, or as an operand of `UNION`, then the enclosing parentheses do not affect the semantics and it denotes a table with one unnamed column.

COUNT (*) denotes the cardinality of the operand table. If the * is replaced by some expression x , then COUNT(x) denotes the number of rows that satisfy the condition x IS NOT NULL. For obvious reasons, that particular aspect of aggregation in SQL has no counterpart in **Tutorial D**.

Example 5.2: Counting the students who have scored more than 50 in some exam

```
( SELECT COUNT(*) FROM
  ( SELECT DISTINCT StudentId
    FROM   EXAM_MARK
    WHERE  Mark > 50 ) AS T )
```

Example 5.2 is a direct translation of the corresponding example in the theory book but it can be abbreviated—and is much more likely to be written—as

```
( SELECT COUNT ( DISTINCT StudentId )
  FROM   EXAM_MARK
 WHERE  Mark > 50 )
```



Hva får egentlig en ingeniør- eller teknologistudent for 300 kroner?

- Medlemskap i en aktiv studentorganisasjon – hele studietiden
- 150 tillitsvalgte studenter som ivaretar dine interesser
- Jobbsøkerkurs
- Gratis PC-forsikring og gode bank- og forsikringstilbud
- Teknisk Ukeblad og NITO Refleks
- Møteplasser på web 2.0

Flere medlemsfordeler og innmelding: www.nito.no/student

Alle som studerer på ingeniør-, bioingeniør-, sivilingeniør eller andre teknologistudier (høgskolekandidat, bachelor eller master) kan bli medlem i NITO.

NITO NORGES STØRSTE ORGANISASJON FOR INGENIØRER OG TEKNOLOGER



Historical Notes

The abbreviated formulation for Example 5.2 has been in SQL from the start. The longer form became available in SQL:1992, when support for “derived tables in the FROM clause” was introduced. Note that the longer form is needed when `SELECT DISTINCT` specifies more than one expression.

The use of parentheses in the `GROUP BY` clause, and support for `GROUP BY ()` in particular, were introduced in SQL:1999. Previously, a `GROUP BY` clause had been required to specify at least one column.

Example 5.3 illustrates the use of `SUM` in SQL

Example 5.3: Adding up all the marks obtained by student S1

```
( SELECT SUM(Mark) FROM EXAM_MARK WHERE StudentId = 'S1' )
```

Effect of NULL

As with `COUNT`, the evaluation of `SUM(x)` ignores rows that satisfy the condition `x IS NULL`, this being a general rule applying to all aggregations in SQL. However, this doesn’t mean that `SUM(x)` is guaranteed never to result in `NULL`, for `NULL` is the result whenever the operand table is empty or `x IS NULL` is *true* for every row. Thus, Example 5.3 results in `NULL` whenever student S1 has taken no exams.

Note that `SUM(x) + SUM(y)` is not in general equivalent to `SUM(x + y)`, because `x + y` evaluates to `NULL` when either `x` evaluates to `NULL` or `y` does. Thus, some values that are included in the summing of `x` and the summing of `y` might be omitted in the summing of `x + y`.

Example 5.4: MAX and MIN

```
( SELECT MAX(Mark) FROM EXAM_MARK WHERE StudentId = 'S1' )  
( SELECT MIN(Mark) FROM EXAM_MARK WHERE StudentId = 'S1' )
```

Example 5.4 needs no further explanation. SQL also has `AVG` for averages. Its counterparts of **Tutorial D**’s aggregate `AND` and `OR` are spelled, respectively, `EVERY` and either `SOME` or `ANY`, but all of these must be used with care because of the consequences of the aforementioned general rule concerning the treatment of `NULL`. For example, if the condition `c` evaluates to `UNKNOWN` for every row of table `t`, or `t` is empty, then `(SELECT EVERY(c) FROM t)` evaluates to `UNKNOWN`, whereas when `t` is empty it really ought to evaluate to `TRUE`. (I hesitate to hazard a guess as to what it should “really” evaluate to when `t` is nonempty but `c` evaluates to `UNKNOWN` in every row.)

At this point the theory book gives Example 5.5 (not repeated here) as an unpleasant way of computing the number of students who sat each exam and shows Figure 5.4, repeated here, as a preferred way of presenting the result of such a computation.

Courseld	n
C1	3
C2	1
C3	1
C4	0

Figure 5.4: How many sat each exam

The example is used as motivation for **Tutorial D**'s `SUMMARIZE` operator but the demonstration that `SUMMARIZE` is indeed a shorthand is rather elaborate, occupying the next three sections of the theory book. The first of those three, Section 5.4, **Relations within a Relation**, describes attributes whose values are relations and shows how such attributes can be obtained using relational extension. Section 5.5, **Using Aggregate Operators with Nested Relations**, then shows how results such as that shown in Figure 5.4 can be obtained using relational operators described in Chapter 4, and then `SUMMARIZE` is introduced in Section 5.6 as a shorthand for obtaining those same results.

As we shall eventually see, SQL has a fairly direct counterpart of `SUMMARIZE BY`, using aggregation in combination with a `GROUP BY` clause. SQL textbooks do not normally teach aggregation and `GROUP BY` along the theory book's lines for teaching `SUMMARIZE`, but I do so here to maintain the parallel structure and also to show how the intermediate results of sections 5.4 and 5.5 can be obtained in modern SQL. The names of these sections are SQL paraphrases of their counterparts in the theory book.

5.4 Tables within a Table

Figure 5.5 here is an exact copy of the one in the theory book and as before it is just an alternative way of representing some of the information conveyed by the tables in Figure 5.1 (but not a recommended database design for that information).

CourseId	ExamResult	
C1	StudentId	Mark
	S1	85
	S2	49
	S4	93
C2	StudentId	Mark
	S1	49
C3	StudentId	Mark
	S3	66
C4	StudentId	Mark

Figure 5.5: Tables within a table

For each course, it shows the exam mark obtained by each student who took the exam for that course. Example 5.6 shows how to obtain this table—let's call it C_ER again—in SQL.

Example 5.6: Obtaining C_ER from COURSE and EXAM_MARK

```
SELECT CourseId,
       CAST (
         TABLE ( SELECT DISTINCT StudentId, Mark
                   FROM   EXAM_MARK AS EM
                   WHERE  EM.CourseId = C.CourseId )
         AS ROW ( StudentId SID, Mark INTEGER ) MULTISET )
  AS ExamResult
FROM   COURSE AS C
```

Explanation 5.6

- The `SELECT` clause operates on each row of the result of the `FROM` clause—i.e., on each row of the `COURSE` table, deriving two columns, `CourseId` and `ExamResult`.
- **`CourseId`** is self-explanatory, merely carrying forward the column values from the column of that name in `COURSE`.
- **`TABLE (SELECT DISTINCT StudentId, Mark FROM EXAM_MARK AS EM WHERE EM.CourseId = C.CourseId)`** denotes a multiset whose elements are rows, obtained by taking the `StudentId` and `Mark` values from those rows of `EXAM_MARK` that match the current row of `COURSE` on `CourseId`. **Note very carefully**, however, that this multiset does not necessarily inherit the column names, `StudentId` and `Mark`, from the table that is the operand to the invocation of `TABLE`. The SQL standard allows the column names to be “implementation-dependent” (i.e., undefined) so long as no two columns have the same name. An implementation that nevertheless carried forward the unique names `StudentId` and `Mark` would be both sensible and conforming, and would obviate the need for the `CAST` invocation explained in the next bullet.

The same multiset would result if the word `DISTINCT` had been omitted, thanks to the `WHERE` condition, but I include it because the example in the theory book uses `COMPOSE`, which is defined as a projection of a join, and SQL’s counterpart of projection uses `SELECT DISTINCT`.

- **`CAST (t AS ROW (StudentId SID, Mark INTEGER) MULTISSET)`**, where *t* is the above `TABLE` expression, addresses the aforementioned possible problem by assigning the required column names. Note that we need to know and write down the declared types of those columns as well as their names. The “type conversion” operator `CAST` is described in Chapter 2, Explanation 2.1a. Here it is being used to convert a value of some incompletely defined multiset type to one whose multiset type is explicitly defined.
- **`AS ExamResult`** then gives the resulting column the name `ExamResult`. Note that here the name comes after `AS` and the expression defining it comes before, in the same style as the use of `AS` to define the range variables `C` and `EM` in the example.

The values for columns such as `ExamResult` in this example have sometimes been referred to informally as *nested tables*, being “tables within a table”, so to speak. Unfortunately, however, they are not actually tables, but rather multisets of rows. Because of that fact, a column such as `ExamResult` cannot appear as an element in a `FROM` clause, so we cannot use it in the way it is used in Example 5.7 in the theory book.

Historical Notes

The `TABLE` operator didn't appear in SQL until SQL:2003 and it remains an optional conformance feature. See Chapter 2, Section 2.6 **What Is a Type?** and Section 2.8, **The Type of a Table**.

It is not surprising that support for nested tables was absent from the early SQL implementations. Codd's *first normal form* (1NF) had been widely understood to be a required property of all relations, not just database relvars, and relation-valued attributes were proscribed under that normal form. The proscription came into question during the 1980s, when the term “not first normal form” was coined, abbreviated to NFNF and thence, somewhat jocularly, to NF².

`CAST` first appeared in SQL:1992.

5.5 Using Aggregation on Nested Tables

Example 5.7 is the most direct translation of its counterpart in the theory book that can be obtained in SQL but it is so over-elaborate that no SQL practitioner would consider using it. It uses the aggregate operator `COUNT` on the table values for column `ExamResult` to obtain the number of students who sat each exam. Unfortunately, as already noted, we cannot operate directly on `ExamResult` as a `FROM` clause element. Instead, we need to use an artifice that is specially devised for the sake of this example.


Skatteetaten



Vil du jobbe i et av landets største IT-miljøer?
Vi skal gjøre det kompliserte enkelt

Skatteetaten tilbyr store fagmiljø og utfordrende oppgaver innen:

- > Systemutvikling
- > Service oriented architecture (SOA)
- > Business intelligence (BI)
- > Testledelse
- > Webutvikling
- > IT sikkerhet
- > Infrastruktur
- > Brukergrensesnitt

For nyutdannede IT-spesialister kan vi tilby et to-årig traineeprogram.

For mer informasjon se skatteetaten.no/jobb

Profesjonell • Nytenkende • Imøtekommende



Example 5.7: How many students sat each exam (not a recommended solution!)

```

WITH C_ER AS (
    SELECT CourseId,
           CAST (
               TABLE ( SELECT DISTINCT StudentId, Mark
                        FROM   EXAM_MARK AS EM
                        WHERE  EM.CourseId = C.CourseId )
               AS ROW ( StudentId SID, Mark INTEGER ) MULTISET )
           AS ExamResult
    FROM   COURSE AS C )
SELECT CourseId, (SELECT COUNT(*)
                  FROM   TABLE(ER(ExamResult)) AS t) AS n
FROM     C_ER

```

Explanation 5.7

- The **WITH** clause, occupying the first nine lines of the example, illustrates SQL's counterpart of **Tutorial D's** construct of the same name. It assigns the name, **C_ER**, to the result of Example 5.6. That name, **C_ER**, is then used in the **FROM** clause of the expression that follows the **WITH** clause. Note that here the name comes before **AS** and the expression defining it comes after. This is consistent with the analogous use of **AS** in **CREATE VIEW** statements, SQL's counterparts of **Tutorial D's** virtual relvar definitions.
- **TABLE (ER (ExamResult))** seems to be the only way of having a multiset valued column operated on as an element of a **FROM** clause—a simple column name is not allowed to appear here. **TABLE (ExamResult)** can't be used either, because when an invocation of **TABLE** appears as a **FROM** clause element, its operand is required to be, specifically, an invocation of a user-defined function. Here I am assuming that **ER** is defined as follows:

```

CREATE FUNCTION ER
( SM ROW ( StudentId SID, Mark INTEGER ) MULTISET )
RETURNS TABLE ( StudentId SID, Mark INTEGER )
RETURN SM ;

```

The type name **TABLE (StudentId SID, Mark INTEGER)** is actually just a synonym for **ROW (StudentId SID, Mark INTEGER) MULTISET**. The misleading synonym is available only in a **RETURNS** clause and not as a parameter type, for example. So **ER** is actually a no-op, returning its input.

- **(SELECT COUNT(*) FROM TABLE(ER(ExamResult)) AS t)** is a scalar subquery, yielding the cardinality of the multiset of rows that is the value of the column `ExamResult` in the current row of `C_ER`. Because we are using the expression to denote a scalar value rather than a table, naming the column would be pointless (apart, perhaps, from injecting a somewhat sarcastic element of purism). As `COUNT(*)` doesn't use a column name, Example 5.7 is valid even if we omit the invocation of `CAST` to assign column names.
- **AS n** then gives the resulting column the name `n`. Note that here the name comes after `AS` and the expression defining it comes before, in the same style as the use of `AS` to define the range variables `C` and `EM` in the example.

So long as `CAST` is used as shown, we could obtain the total marks for each exam in similar fashion, using `SUM(Mark) AS TotalMarks`. However, this gives `NULL`, instead of zero, for the courses whose exams nobody sat. That problem can be addressed by using `COALESCE`, as shown in Example 5.7a.

Example 5.7a: Give the total of marks for each exam (still not a recommended solution)

```
WITH C_ER AS (
    SELECT CourseId,
           CAST (
               TABLE ( SELECT DISTINCT StudentId, Mark
                       FROM   EXAM_MARK AS EM
                       WHERE  EM.CourseId = C.CourseId )
               AS ROW ( StudentId SID, Mark INTEGER ) MULTISET )
           AS ExamResult
    FROM   COURSE AS C )
SELECT CourseId, COALESCE (
                                (SELECT SUM(Mark)
                                FROM   TABLE(ER(ExamResult)) AS t), 0)
AS n
FROM     C_ER
```

Explanation 5.7a

- **COALESCE((SELECT SUM(Mark) FROM TABLE(ER(ExamResult)) AS t), 0)** yields the value of the scalar subquery whenever `ExamResult` is nonempty, otherwise zero. SQL's `COALESCE` is an n -adic operator that takes a commalist of expressions of the same declared type and yields the result of the first of those expressions, in the order in which they are written, that does not evaluate to `NULL`. When there is no such operand, it yields `NULL` anyway, of course.

Notice that SQL does not follow the rule given in the theory book whereby aggregation over the empty relation yields the identity value for the basis operator (when such a value exists), in this case addition, whose identity value is zero.

Now, as I remarked already, examples 5.7 and 5.7a are not recommended ways to obtain the required result. Example 5.7b shows how 5.7a can be condensed into a much more concise solution.

Example 5.7b: Give the total of marks for each exam (simplified solution)

```
SELECT CourseId,  
       COALESCE ( ( SELECT SUM(Mark)  
                   FROM   EXAM_MARK AS EM  
                   WHERE  EM.CourseId = C.CourseId ),  
                 0 ) AS TotalMark  
FROM   COURSE AS C
```

Explanation 5.7b

- The first operand to the invocation of `COALESCE` is a scalar subquery similar to those shown in Section 5.3, **Aggregate Operators**. Recall the need to enclose the `SELECT` expression in parentheses, without which it would denote a table rather than a number.

Historical Notes

The SQL standard's `WITH` clause first appeared in SQL:1999. It remains an optional conformance feature. Moreover, even if an implementation does support it, it is further optional as to whether it is permitted to appear in a subquery.

Scalar subqueries have been in SQL from the beginning but were originally allowed to appear only as the second operand of a comparison, and thus could not appear in the `SELECT` clause. Moreover, they were subject to various arbitrary restrictions, such as not being allowed to contain `UNION`, `GROUP BY`, or `HAVING`. The restrictions were lifted in SQL:1992 but implementations that retained them were still able to claim the minimum level of conformance until the appearance of SQL:1999.

The `COALESCE` operator was added to the language in SQL:1992. It ceased to be optional in SQL:1999.

If we need the average mark for each exam we will have to avoid zero-divides by excluding those which no students sat. As in the theory book we can do that by homing in on just those `CourseId` values that appear in `EXAM_MARK`, as shown in Example 5.8, the differences from Example 5.7b being shown in bold.

Example 5.8: Average mark per exam

```

SELECT CourseId,
      ( SELECT AVG (Mark)
        FROM   EXAM_MARK AS EM
        WHERE  EM.CourseId = C.CourseId ) AS AvgMark
FROM   EXAM_MARK AS C

```

Now, the theory book goes on from here to describe two varieties of the relational summarization operator—`SUMMARIZE PER` and `SUMMARIZE BY` in **Tutorial D**—providing useful shorthands for expressions like Example 5.7. SQL has no such operators but it does provide a useful shorthand for cases that in **Tutorial D** can be formulated using `SUMMARIZE BY`, as we shall see in the next section. As with `SUMMARIZE`, this shorthand also allows multiple aggregations to be specified on the same table without repeating the expression denoting that table. Example 5.8a shows how this repetition problem arises if we followed the style of Example 5.8 to obtain both the average mark and the total, this time ignoring exams that nobody sat.

Example 5.8a: Average and total mark per exam (not a recommended solution!)

```

SELECT CourseId,
      ( SELECT AVG (Mark)
        FROM   EXAM_MARK AS EM
        WHERE  EM.CourseId = C.CourseId ) AS AvgMark,
      ( SELECT SUM (Mark)
        FROM   EXAM_MARK AS EM
        WHERE  EM.CourseId = C.CourseId ) AS TotalMark
FROM   EXAM_MARK AS C

```

5.6 Summarization in SQL

Example 5.9 in the theory book uses **Tutorial D**'s `SUMMARIZE PER` to give the same result as Example 5.7. Because SQL has no direct counterpart of `SUMMARIZE PER`, here I need to go straight to Example 5.11 to show how `SUMMARIZE BY` invocations can be simulated in SQL. Then I can show how Example 5.9 could be translated in SQL. Example 5.11 introduces us to SQL's `GROUP BY` clause, this being its direct counterpart of **Tutorial D**'s `BY` specification in `SUMMARIZE BY`.

Example 5.11: Average mark for each exam, using `GROUP BY` (recommended solution)

```

SELECT   CourseId, AVG(mark) AS AvgMark
FROM     EXAM_MARK
GROUP BY CourseId

```

The `GROUP BY` clause is not to be confused with **Tutorial D**'s `GROUP` operator. Actually, `GROUP BY CourseId` can be considered to have the same effect as `GROUP{ALL BUT CourseId}` in **Tutorial D**, in which case `AVG (Mark)` then operates on the nested tables produced by the `GROUP BY` clause. However, the SQL standard and most SQL textbooks do not define `GROUP BY` in such terms. Rather, they introduce the notion of partitioning the body of a table into *groups* and refer to such a partitioned table as a *grouped table*. When a `SELECT` expression includes a `GROUP BY` clause, each `SELECT` clause element must specify a column that is functionally dependent on the `GROUP BY` columns, thus reducing each group to a single row. (Functional dependence is taught in Chapter 7 of the theory book. It should be clear to you that the FD $\{CourseId\} \rightarrow \{CourseId, AvgMark\}$ always holds in the result of Example 5.11.)

We can return to the theory book's Example 5.9 now because we can use `GROUP BY` to obtain part of the required result and an outer join (see Chapter 4, Example 4.1e) in conjunction with `COALESCE` to complete it. The example as given could be addressed in similar fashion to Example 5.7 but the method shown in Example 5.9 is likely to be preferred when more than one aggregation is to be specified on the same table.

Example 5.9: How many students sat each exam,
using `GROUP BY`, `NATURAL LEFT JOIN`, and `COALESCE`

```
SELECT CourseId, COALESCE(n, 0) AS n
FROM   COURSE NATURAL LEFT JOIN
      ( SELECT CourseId, COUNT(*) AS n
        FROM   EXAM_MARK
        GROUP BY CourseId ) AS T
```

Explanation 5.9

- **NATURAL JOIN** is described in Chapter 4, Section 4.1. Note, however, that the use of `LEFT` makes this an outer join, whereas Codd's term *natural join* referred to the "inner" variety only.
- **LEFT** specifies that each unmatched row in the first join operand, `COURSE`, is to be extended with `NULL` for the column `n`.
- **COALESCE (n, 0) AS n** effectively replaces those appearances of `NULL` by the correct value, 0.

(I give no counterpart for Example 5.10 in the theory book because it merely shows how **Tutorial D**'s `SUMMARIZE BY` is just a shorthand for certain special cases of `SUMMARIZE PER`.)

Historical Notes

In the final section of Chapter 4, **Concluding Remarks**, I mentioned a claim that “early versions” of SQL were not relationally complete, for reasons beyond the “small” matter of failing to recognize the existence of relations of degree zero (or tables with no columns). I can now give the rationale for that claim. Example 5.9 uses a `SELECT` expression in its `FROM` clause, which was not supported by the international standard until SQL:1992 appeared. Without such support it is not always possible for the table resulting from such an expression to be joined with another table. In particular, it is not possible when a join is required of two tables that are both obtained by use of `SELECT ... FROM ... GROUP BY`.

5.7 Grouping and Ungrouping in SQL

Example 5.6a is derived from Example 5.6 by specifying `EXAM_MARK` in place of `COURSE` in the main `FROM` clause.

Example 5.6a: Obtaining `C_ER2` from `EXAM_MARK`

```
SELECT  CourseId,
        CAST (
            TABLE ( SELECT DISTINCT StudentId, Mark
                     FROM   EXAM_MARK AS EM2
                     WHERE  EM1.CourseId = EM2.CourseId )
            AS ROW ( StudentId SID, Mark INTEGER ) MULTISET )
        AS ExamResult
FROM    EXAM_MARK AS EM
```

Figure 5.4 shows the result, named `C_ER2` for convenience. It differs from the `C_ER` of Figure 5.3 only in the absence of a row for course `C4`, whose exam nobody sat.

Courseld	ExamResult	
C1	StudentId	Mark
	S1	85
	S2	49
	S4	93
C2	StudentId	Mark
	S1	49
C3	StudentId	Mark
	S3	66

Figure 5.4: Intermediate result C_ER2 from Example 5.6a



OLJE- OG ENERGIDEPARTEMENTET



Er du full av energi?

Olje- og energidepartementets hovedoppgave er å tilrettelegge for en samordnet og helhetlig energipolitikk. Vårt overordnede mål er å sikre høy verdiskapning gjennom effektiv og miljøvennlig forvaltning av energiressursene.

Vi vet at den viktigste kilden til læring etter studiene er arbeidssituasjonen. Hos oss får du:

- Innsikt i olje- og energisektoren og dens økende betydning for norsk økonomi
- Utforme fremtidens energipolitikk
- Se det politiske systemet fra innsiden
- Høy kompetanse på et saksfelt, men også et unikt overblikk over den generelle samfunnsutviklingen
- Raskt ansvar for store og utfordrende oppgaver
- Mulighet til å arbeide med internasjonale spørsmål i en næring der Norge er en betydelig aktør

Vi rekrutterer sivil- og samfunnsøkonomer, jurister og samfunnsvitere fra universiteter og høyskoler.

www.regjeringen.no/oed



Se ledige stillinger her

www.jobb.dep.no/oed



The theory book gives Example 5.12 to illustrate use of the operator `GROUP` as a shorthand for obtaining such results. SQL has no direct counterpart of `GROUP` but Example 5.12 shows how to use `GROUP BY` to obtain an alternative formulation to produce `C_ER2` that is more concise than Example 5.8a.

Example 5.12: Using `GROUP BY` and `COLLECT` to obtain `C_ER2`

```
SELECT      CourseId,
           CAST (
               COLLECT (ROW (StudentId, Mark))
               AS ROW ( StudentId SID, Mark INTEGER ) MULTISET )
           AS ExamResult
FROM        EXAM_MARK
GROUP BY    CourseId
```

Explanation 5.12

- **`ROW (StudentId, Mark)`** forms the row consisting of the `StudentId` and `Mark` values of the current row of `EXAM_MARK`, in that order. The two fields of this row are unnamed.
- **`COLLECT (ROW (StudentId, Mark))`** collects together as a multiset all of those rows that are derived `EXAM_MARK` rows having the same `CourseId` value. In fact it is shorthand for `FUSION (ROW (StudentId, Mark) MULTISET)`, where `FUSION` is SQL's nearest counterpart of **Tutorial D's** aggregate `UNION`. For each value of its operand, `COLLECT` derives the multiset containing just that value, and returns the `FUSION` (see next bullet) of all the multisets thus formed.
- `FUSION` is aggregate multiset union (`UNION ALL`), not `UNION` per se. In general the same value (in our example, a row) might appear more than once in the result of a `COLLECT` invocation. Fortunately, that won't happen here because the same `StudentId, Mark` combination cannot appear along with the same `CourseId` in more than one row of `EXAM_MARK`, so `DISTINCT` could be omitted.

Tutorial D's aggregate operator `UNION` is not mentioned in the theory book. It is used for taking the union of the relations appearing as values of a specified attribute in the relation operand of that aggregate operator.

- **`CAST (m AS ROW (StudentId SID, Mark INTEGER) MULTISET)`**, where `m` is the above `COLLECT` expression, names the columns of the nested table, `ExamResult`. Note the need to spell out the entire declared type of `ExamResult`, even though it differs from that of the `COLLECT` expression only in the names of the two columns.

By the way, the following table expression:

```
SELECT *
FROM ( FUSION ( TABLE ( VALUES ( StudentId, Mark ) ) ) )
      AS T(StudentId, Mark)
```

denotes the same value as the given `CAST` expression but unfortunately it cannot be used as an alternative for the very reason that it is a table expression—table expressions are not permitted as `SELECT` clause elements. We could enclose it in parentheses following the word `TABLE`, but then, as I have already explained, we have no guarantee that the column names would be propagated to the result and so we might have to use `CAST` again after all!

That table expression is rather convoluted. You might prefer not to be given an explanation for it but here it is anyway:

`(StudentId, Mark)` denotes the row consisting of the `StudentId` value followed by the `Mark` value. Putting `VALUES` in front of it makes it into table expression denoting the table containing just that row. Putting `TABLE` in front of that makes it into a multiset expression, as required by `FUSION`. Although table expressions are not permitted as `SELECT` clause elements, multiset expressions *are* permitted as `FROM` clause elements, allowing us to enclose the `FUSION` invocation in `SELECT * FROM (...) AS T(StudentId, Mark)` to make sure we have the required column names.

In **Tutorial D**, the inverse operator of `GROUP` is `UNGROUP`. SQL has an operator, `UNNEST`, that can be used for similar purposes, but its method of invocation is somewhat peculiar, as Example 5.13 shows, and it can be used only to specify a `FROM` clause element.

Example 5.13: Inverse of Example 5.12, using `UNNEST`

```
SELECT DISTINCT * FROM C_ER2, UNNEST ( ExamResult ) AS M
```

The name `C_ER2` could be defined using a `WITH` clause, as in Example 5.7a. Notice how the second element of the `FROM` clause has to be reevaluated for each row of `C_ER2`, whereas each `FROM` clause element is normally evaluated just once because its value does not vary from row to row of previous elements. The column reference `ExamResult` is a reference to the column of that name in `C_ER2` and is permitted only because `C_ER2` is specified *before* `UNNEST (ExamResult)` in the `FROM` clause—a switching of these two `FROM` clause elements would result in a syntax error.

Effect of NULL

In Example 5.13, ExamResult is a column of type ROW (StudentId SID, Mark INTEGER) MULTiset). In C_ER2 NULL cannot appear in place of a value for that column, but in general NULL can appear in place of a value for a column of some multiset type. So we need to know what happens when NULL is given as the argument to an invocation of UNNEST. At the time of writing (in 2012), the SQL standard appears to be silent on that issue. One would expect it to give rise to an exception condition.

Note also that NULL might in general appear as an element of a multiset whose element type is a ROW type, though again this cannot arise in C_ER2, assuming that C_ER2 is derived as shown in Example 5.12.

Historical Note

UNNEST first appeared in SQL:1999, though in that edition it was used only with arrays, not with multisets. COLLECT and FUSION first appeared in SQL:2003, along with INTERSECTION, for computing an aggregate intersection of multisets (and not to be confused with the table operator INTERSECT). They are all optional conformance features.



HELT GRATIS!

S for Skikk & Bank

DU FÅR BOKA
HOS DNB

S for Skikk & Bank

En bok om ting som er greit å vite når du har flyttet hjemmefra.

dnb.no

DNB

Bank fra A til Å



Now, the theory book at this point observes that the cardinality of the result of Example 5.13 is equal to the sum of the cardinalities of the `ExamResult` values in `C_ER2`. It then poses the following question as an *exercise for the reader*: Is it *always* the case that the cardinality of an ungrouping is equal to the sum of the cardinalities of the relations the operand relation is being ungrouped on? Although I didn't need to include `DISTINCT` in this example, the fact that I decided to do so anyway gives you a broad hint as to the correct answer to that question. Can you think of an example where `DISTINCT` would be required to avoid duplicates?

5.8 Wrapping and unwrapping in SQL

The theory book describes operators `WRAP` and `UNWRAP` in connection with attributes whose declared types are tuple types. Example 5.14 in that book shows how extension and projection can be used to replace a given set of attribute values in each tuple of a given relation by a single tuple consisting of those values; the next example then illustrates the use of `WRAP` as a convenient shorthand for the same purpose (on the admittedly rare occasions on which it is likely to arise in practice).

The effect of Example 5.14 can be obtained in SQL but note that one needs to write down not only the names of the columns being wrapped but also the names and declared types of the columns not being wrapped.

Example 5.14: Collecting column values together

```
SELECT Name, Phone, Email,
       CAST ( ROW ( House, Street, City, Zip ) AS
             ROW ( House VARCHAR(100), Street VARCHAR(100),
                  City VARCHAR(100), Zip VARCHAR(10) ) )
       AS Address
FROM   CONTACT_INFO
```

As before, we need to use `CAST` because the result of an invocation of `ROW` has unnamed fields. The example assumes, therefore, a definition such as the following for the base table `CONTACT_INFO`:

```
CREATE TABLE CONTACT_INFO ( Name VARCHAR(100) PRIMARY KEY,
                             Phone VARCHAR(15) NOT NULL,
                             Email VARCHAR(50) NOT NULL,
                             House VARCHAR(100) NOT NULL,
                             Street VARCHAR(100) NOT NULL,
                             City VARCHAR(100) NOT NULL,
                             Zip VARCHAR(10) NOT NULL,
                             ) ;
```

SQL has no shorthand similar to `WRAP`, nor for `UNWRAP`. Example 5.15 here shows how unwrapping can be done in longhand in SQL.

Download free eBooks at bookboon.com

Example 5.15: Unwrapping in SQL (inverse of Example 5.14)

Letting `CONTACT_INFO_WRAPPED` denote the result of Example 5.14:

```
SELECT Name,
       Address.House as House,
       Address.Street as Street,
       Address.City as City,
       Address.Zip as Zip
FROM   CONTACT_INFO_WRAPPED
```

`Address.House` in this example is equivalent to **Tutorial D**'s `House FROM Address`. The use of a dot here is consistent with its use with range variables—recall that a range variable also denotes a row, “ranging” over the rows of a table.

5.9 Table Comparison

The theory book includes the following definitions for relation comparisons in **Tutorial D**:

Let $r1$ and $r2$ be relations having the same heading. Then:

- $r1 \subseteq r2$ is *true* if every tuple of $r1$ is also a tuple of $r2$, otherwise *false*.
- $r1 \supseteq r2$ is equivalent to $r2 \subseteq r1$
- $r1 = r2$ is equivalent to $r1 \subseteq r2$ AND $r2 \subseteq r1$

The question arises as to whether SQL tables can be similarly compared. SQL does not have direct counterparts of \subseteq and \supseteq . It does of course have $=$, but table expressions cannot be used as comparands. However, as we have seen in Examples 5.6 et seq., the operator `TABLE` has been available since SQL:2003 to derive from a given table expression a value of a multiset type whose element type is a row type. In other words, $(\text{SELECT } * \text{ FROM } t1) = (\text{SELECT } * \text{ FROM } t2)$ is illegal but we can obtain the required effect by writing `TABLE (SELECT * FROM t1) = TABLE (SELECT * FROM t2)`. So, to compare two tables, we have to use an operator named `TABLE` to “convert” them from tables into multisets of rows!

To test for every row of $t1$ being also a row of $t2$ we could write, for example, `NOT EXISTS (SELECT * FROM t1 EXCEPT SELECT * FROM t2)`. In fact, SQL's `NOT EXISTS` is an exact counterpart of **Tutorial D**'s `IS_EMPTY` operator. However, note carefully that the case where every row in $t1$ appears in $t2$ and every row of $t2$ appears in $t1$ does not guarantee that $t1$ and $t2$ are the same table. Row r might appear twice in $t1$ but only once in $t2$, for example.

You should now be able to write SQL counterparts for the theory book's Examples 5.16 and 5.17, so I leave those as exercises for the reader.

5.10 Other Operators on Tables and Rows

Section 5.10 in the theory book covers some of **Tutorial D**'s additional operators involving relations and tuples. If only for the sake of completeness, we need to look for SQL counterparts of these.

Row Membership Test: We have already seen, in Section 5.2, SQL's IN operator, for **Tutorial D**'s \in (spelled as IN in *Rel*).

Row Extraction

For **Tutorial D**'s TUPLE FROM *r*, SQL has *row subqueries*. These are just like scalar subqueries (see Section 5.3) except that they may specify more than one column. For example, when appearing in a suitable context, the expression

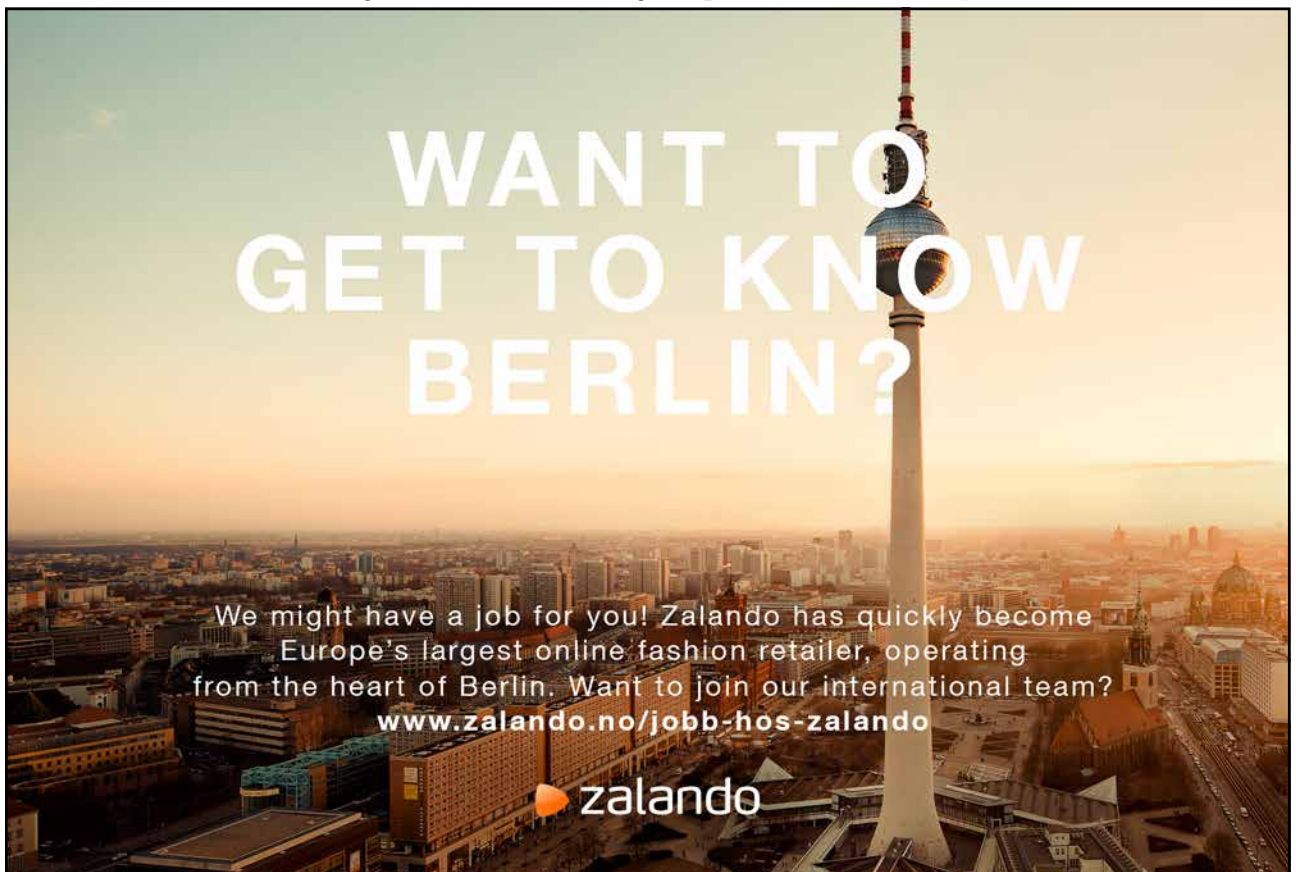
```
( SELECT * FROM COURSE WHERE CourseId = CID('C1') )
```

yields the row denoted by

```
CAST ( ROW ( 'C1', 'Database' ) AS
        ROW ( CourseId CID, Title VARCHAR(100) ) )
```

A row subquery whose table operand is empty yields a row in which every field is NULL.

Field Extraction: For extracting a field from a row, using dot qualification, see Example 5.15 in Section 5.8.



Row Counterparts of Table Operators

SQL does not have counterparts of **Tutorial D**'s tuple rename, tuple projection, tuple extension, tuple join and tuple compose. To obtain the same effects as these operators on row r , one has first to derive the table t consisting of just r , then apply the SQL counterpart of the corresponding relational operator on t , putting parentheses around the table expression so that, so long as the context is appropriate, it becomes a row subquery.

For example, if r has fields named a , b , and c , we can simulate a tuple renaming of a tuple projection to obtain the row consisting of just a and b , with b renamed to x , by (SELECT a , x FROM VALUES (r) AS $t(a, x, c)$). The snag here is that the columns of a VALUES expression have implementation-dependent names, so we cannot rely on the field names of r being propagated to the table. We therefore have to specify the names in parentheses after the range variable name, t . At least that gives us the slight short cut of renaming b as x on the fly, so to speak.

EXERCISES

1. Does SQL have a counterpart of $r1$ COMPOSE $r2$ when $r1$ and $r2$ have identical headings? If so, what is it, in general? If not, why not?
2. Write an SQL expression that is equivalent to Example 5.9, repeated below, but does not use a SELECT expression in a FROM clause.

```
SELECT  CourseId, COALESCE(n, 0) AS n
FROM    COURSE LEFT NATURAL JOIN
        ( SELECT  CourseId, COUNT(*) AS n
          FROM    EXAM_MARK
          GROUP BY CourseId ) AS T
```

3. Write an SQL expression that is equivalent to the example below but does not use a SELECT expression in a FROM clause.

```
SELECT  CourseId, Title, AvgMark
FROM    COURSE NATURAL JOIN
        ( SELECT  CourseId, AVG(Mark) AS AvgMark
          FROM    EXAM_MARK
          GROUP BY CourseId ) AS T
```

4. In connection with Example 5.13, can you give an example where the same row might appear more than once in the result if DISTINCT is omitted? If so, give it; otherwise explain.

5. Using the suppliers-and-parts database shown in Figure 4.13, write SQL expressions for the following queries:

- a) Get the total number of parts supplied by supplier S1.
- b) Get supplier numbers for suppliers whose city is first in the alphabetic list of such cities.
- c) Get part numbers for parts supplied by all suppliers in London.
- d) Get supplier numbers and names for suppliers who supply all the purple parts.
- e) Get all pairs of supplier numbers, S_x and S_y say, such that S_x and S_y supply exactly the same set of parts each.
- f) Write a truth-valued expression to determine whether all supplier names are unique in S .
- g) Write a truth-valued expression to determine whether all part numbers appearing in SP also appear in P .

6. Give SQL counterparts of the theory book's Examples 5.16 and 5.17.

7. **Tutorial D's** `TUPLE` operator takes a commalist of expressions, each one paired with an attribute name. By contrast, SQL's `ROW` operator takes a commalist of expressions without accompanying field names. What are the advantages and disadvantages of SQL's approach?

8. Distinguish between SQL's table types and its multiset types whose element types are row types.

6 Constraints and Updating

6.1 Introduction

As in the theory book, this chapter deals with database constraints, not to be confused with type constraints (not supported in SQL) and SQL's so-called domain constraints discussed in Chapter 2.

In Chapter 1, Example 1.3, you saw a simple example of a database constraint declaration expressed in SQL, repeated here as Example 6.1 (though now referencing `IS_ENROLLED_ON` rather than `ENROLMENT`).

Example 6.1: Declaring an integrity constraint.

```
CREATE ASSERTION MAX_ENROLMENTS
  CHECK ( ( SELECT COUNT(*) FROM IS_ENROLLED_ON ) <= 20000 ) ;
```

`CREATE ASSERTION` is SQL's counterpart of **Tutorial D's** `CONSTRAINT`, but it is an optional conformance feature that first appeared in SQL:1992 and very few SQL implementations (at the time of writing in 2012) support it.

Without `CREATE ASSERTION`, one might attempt to implement the required constraint along the lines of Example 6.1a.

Example 6.1a: Alternative formulation for `MAX_ENROLMENTS`

```
ALTER TABLE IS_ENROLLED_ON
ADD CONSTRAINT MAX_ENROLMENTS
  CHECK ( ( SELECT COUNT(*) FROM IS_ENROLLED_ON ) <= 20000 ) ;
```

Explanation 6.1a

- **ALTER TABLE IS_ENROLLED_ON** announces that an alteration to the definition of base table IS_ENROLLED_ON is being specified.
- **ADD CONSTRAINT MAX_ENROLMENTS** states that the alteration in question is the addition of something SQL calls a *table constraint*, and its name is MAX_ENROLMENTS. A table constraint is a condition that is required to be satisfied by every row appearing in the base table for which that constraint is defined. Thus, in general, it is an open expression of the kind that can appear as the condition of a WHERE clause. In this example the condition is in fact a closed expression—it contains no reference to a column of IS_ENROLLED_ON—and thus if it is satisfied by one row of that table, then it is satisfied by all of them.
- The last line is exactly as written in Example 6.1, but there’s a subtle difference in meaning. Because a table constraint is one that must be satisfied by every row of the applicable table, it is always satisfied when the applicable table is empty—there is no row for which the constraint fails. In the case at hand, this is not a problem, because obviously, when IS_ENROLLED_ON is empty, then its cardinality—zero—does not exceed 20,000. However, suppose a constraint was required to the effect that IS_ENROLLED_ON must never be empty. That could be achieved by changing `<= 20000` to `> 0` in Example 6.1, but the same change to Example 6.1a would be ineffectual: when IS_ENROLLED_ON is empty, it contains no row that fails to satisfy the constraint. That is why SQL is incomplete with respect to database integrity when support for CREATE ASSERTION is absent.

Now, I wrote, “one might attempt to implement the required constraint” this way, suggesting that it might not be such a good idea after all. Even though it does have the desired effect in the example at hand, the fact that a table constraint is one that is required to be satisfied by each row in the relevant table means that the DBMS is very likely to evaluate it for each row that is added to or updated in the table, whereas of course it needs to be evaluated just once per update operation that affects IS_ENROLLED_ON. This is why **Tutorial D** has no counterpart of SQL’s table constraints. They provide a useful shorthand for certain special cases (like the NOT NULL constraint on a column, or a check for a column having nonnegative values only, for example) but they give rise to traps when used inappropriately. A favourite example is `CHECK (EXISTS (SELECT * FROM t))`, as a table constraint on table *t*. It is satisfied even when *t* is empty!

Unfortunately (or fortunately?), Example 6.1a is in any case somewhat hypothetical, because the condition contains a table expression—a subquery. The appearance of a subquery in a table constraint remains an optional conformance feature and to this day many implementations fail to support it—and of those that do support it, at least one that does so fails to enforce such constraints at all times.

One of my reviewers (Erwin Smout) showed me a possible workaround for use when the “no subqueries in table constraints” restriction is in force. Example 6.1b applies this “hack” to Example 6.1a.

Example 6.1b: Workaround for when subqueries not permitted in CHECK constraints

```
CREATE FUNCTION NO_MORE_THAN_20000_ENROLMENTS ( )
    RETURNS BOOLEAN ;
    RETURN ( SELECT COUNT(*) FROM IS_ENROLLED_ON ) <= 20000 ;

ALTER TABLE IS_ENROLLED_ON
ADD CONSTRAINT MAX_ENROLMENTS
    CHECK ( NO_MORE_THAN_20000_ENROLMENTS ( ) ) ;
```

Caveat lector: You are strongly advised to complete Exercise 2 in this chapter’s Exercises section before you commit to using this workaround in earnest.

“Not Enforced” Table Constraints

A constraint that is not enforced is not really a constraint within the meaning of the act, but SQL does have such a concept and it needs to be mentioned here. With the exception of UNIQUE and PRIMARY KEY specifications (see Section 6.4, the subsection headed **Keys**), a table constraint can be declared as either ENFORCED (the default option) or NOT ENFORCED. The “enforcement characteristic” of such a constraint can be changed by means of an ALTER TABLE statement. For example, the table constraint MAX_ENROLMENTS becomes not enforced by execution of the statement ALTER TABLE ALTER CONSTRAINT MAX_ENROLMENTS NOT ENFORCED. The immediate effect is the same (ignoring effects on the catalog) as if ALTER TABLE DROP CONSTRAINT MAX_ENROLMENTS had been given. However, it’s easier to reinstate the constraint using ALTER TABLE ALTER CONSTRAINT MAX_ENROLMENTS ENFORCED than to repeat the whole of Example 6.1a.

NOT ENFORCED cannot be specified for domain constraints or assertions, so use of CREATE ASSERTION should really be the preferred method of declaring a constraint, but unfortunately that option is not widely available.

Historical Note and Comments

[NOT] ENFORCED first arrived in SQL:2007. It remains an optional conformance feature. At first sight it seems to be rather a strange feature, but, assuming there is a genuine requirement for it, one could observe that the effect of switching a table constraint between ENFORCED and NOT ENFORCED can be obtained less conveniently by using ALTER TABLE/DROP CONSTRAINT and ALTER TABLE/ADD CONSTRAINT.

One possible motivation for this feature lies in performance considerations. The feasibility of addressing some integrity requirements decreases with database size and frequency of updates. Rather than leave the database exposed permanently to the possibility of becoming inconsistent, one could declare a “not enforced” constraint and switch it to being enforced at a convenient time. Of course, if the constraint is then found to be violated, then some ad hoc intervention will be needed to address the problem. This approach to maintaining integrity is clearly far from perfect but is perhaps better than nothing in circumstances that offer no practical alternative.

All of that having been said, why the feature is not available with key constraints and constraints declared by `CREATE ASSERTION` is a mystery to this writer.

It might seem that feature is restricted to constraints that have been explicitly named. However, the SQL standard specifies that an implementation-dependent constraint name is given by default. In that case, the unique name assigned by the system will show up in the catalog and could then be used in an `ALTER TABLE/ALTER CONSTRAINT` or `ALTER TABLE/DROP CONSTRAINT` statement.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"

Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



6.2 A Closer Look at Constraints and Consistency

Effects of NULL

Section 6.2 in the theory book starts with

A constraint is defined by a truth-valued expression, such as a comparison. A database constraint is defined by a truth-valued expression that references the database. To be precise, the expression defines a *condition* that must be *satisfied* by the database at all times.

and goes on to justify the use of the term *satisfied*. Unfortunately, SQL has two definitions of this term. A row satisfies a condition in a WHERE clause only when the condition evaluates to TRUE, but it satisfies a table constraint when the condition evaluates to either TRUE or UNKNOWN. Thus, `SELECT * FROM T WHERE c` might yield an empty table even when *c* is the condition specified in some table constraint for T and T itself is far from empty.

When Are Constraints Checked?

Under the model described in the theory book, constraints are conceptually checked at all *statement boundaries* (and only at statement boundaries). By default the same is true of SQL. However, SQL does not support the “multiple assignment” concept, described in the theory book, for database updates. For that reason it has to include an alternative method of addressing the problems that multiple assignment addresses. SQL does so by allowing the checking of specified constraints to be temporarily deferred and reinstated later—but never across a *transaction boundary*. As a result, it is possible for the database to appear to be inconsistent, but only to the user whose as yet uncommitted transaction has given rise to that state of affairs.

As a consequence of deferred constraint checking, SQL code that depends on consistency with declared constraints is obviously exposed to that assumption of consistency being false when the code is executed while checking is deferred. For example, the table expression `SELECT Name FROM IS_CALLED WHERE StudentId = 'S1'` might be expected never to result in a table containing more than one row, thanks to the key constraint applying to IS_CALLED; thus it might be used in a scalar subquery. However, if the checking of that key constraint is temporarily deferred and two or more rows with StudentId equal to 'S1' temporarily appear in that table, then the scalar subquery will give rise to a run-time exception. Fortunately, SQL does allow a constraint to be declared as `NOT DEFERRABLE`, and that is the default option.

Historical Note

Deferred constraint checking first arrived in SQL:1992. It remains an optional conformance feature.

6.3 Expressing Constraint Conditions


Use of Table Expressions

With the exception of key constraints, the examples in the theory book all explicitly reference at least one relvar and thus involve invocations of relational operators or aggregate operators. Assuming support for `CREATE ASSERTION`, we can always derive SQL counterparts of these examples using table expressions and truth-valued operators, but when that assumption does not hold we need to look for alternative solutions using table constraints. In most cases these will entail the use of subqueries and even that technique is prohibited by many implementations. In some cases special syntactic constructs are available, as we shall see, but there are several for which no SQL solution is available unless the implementation supports `CREATE ASSERTION` or subqueries in table constraints.

Now, the reason usually given for lack of support for subqueries in constraints is that in general such expressions can require the DBMS to examine the entire content of possibly very large tables. If database updates are expected to occur frequently—and are perhaps required to occur very frequently indeed—then declaration of such constraints would give rise to an intolerable slowing down of the updating process. Of course this is an extremely valid concern and we have to admit that integrity might occasionally have to be compromised for performance reasons, but consider the user with a small database that is subject to comparatively infrequent updating but nevertheless has strong integrity requirements. Might not such a user feel unfairly treated by a system that prohibits the declaration of required constraints? Defenders of the status quo respond to this argument by holding that language constructs that can give rise to disappointment for performance reasons, to such an extent as to militate against their use in common practical situations, should be banned. But sometimes users resort to implementing constraints, as best they can, in application code when they wish to enforce a constraint that is not supported by the DBMS but nevertheless does not adversely impair performance. The DBMS could almost certainly enforce such constraints much more efficiently *and much more reliably*. We can also point to various other SQL constructs that might be subject to similar concerns but are supported nonetheless. For example, if tables T1, T2, and T3 each contain 100,000 rows, then `SELECT * FROM T1, T2, T3`, when evaluated, delivers a table containing a quadrillion rows.

Procedural Constraint Enforcement (Triggers)

SQL has an alternative method of addressing database integrity, involving event-driven procedural code. The special procedures that can be used for this purpose are called *triggers* and the events that activate them are specified update operations. For example, suppose it is required for every row in `IS_CALLED` to have a matching row on `StudentId` in `IS_ENROLLED_ON`, enforcing a business rule to the effect that every registered student must be enrolled on at least one course. Then a triggered procedure might be activated every time `INSERT` is used to add a row to `IS_CALLED`, checking to see if a matching row exists in `IS_ENROLLED_ON` and raising an exception if there isn't one. But that wouldn't be sufficient to address the requirement. Further triggers would be needed, activated by `UPDATE` statements on `IS_CALLED` and `IS_ENROLLED_ON` that cause changes to `StudentId` values in either of those tables, and by `DELETE` statements on `IS_ENROLLED_ON`. As this simple example demonstrates, use of triggered procedures for constraint enforcement can be complicated and error-prone. As one practitioner told me, "It quickly gets *so* complicated that it's almost impossible for a human *not* to make errors..., and even when you're not facing a 'complicated' case, the work to be done is tedious and boring". The subject is beyond the scope of this book but is dealt with at length and in meticulous detail by the authors of reference [13].



WHILE YOU WERE SLEEPING...

www.fuqua.duke.edu/whileyouweresleeping

DUKE
THE FUQUA
SCHOOL
OF BUSINESS



Use of COUNT and NOT EXISTS

The theory book describes and discusses various general methods of expressing constraints, eventually noting that support for “=” with relation operands is sufficient for completeness. It also notes that every constraint can be expressed as an invocation of IS_EMPTY, where $IS_EMPTY(r)$ is equivalent to $r\{\} = TABLE_DUM$. First, though, it gives Example 6.2, showing how to use COUNT to test a relation for emptiness. Example 6.2 here is a direct translation of that one into SQL.

Example 6.2: Testing for absence of counterexamples.

```
CREATE ASSERTION Must_be_enrolled_to_take_exam
CHECK ( ( SELECT COUNT(*)
          FROM EXAM_MARK
          WHERE ( Student_Id, CourseId ) NOT IN
                ( SELECT Student_Id, CourseId
                  FROM IS_ENROLLED_ON ) )
        = 0 ) ;
```

Of course, counting all the rows is rather excessive when it is sufficient just to see if the table contains anything at all. Examples 6.3 and 6.3a illustrate the use of NOT EXISTS, SQL’s counterpart of Tutorial D’s IS_EMPTY operator. Example 6.3a shows how to express the constraint as a table constraint in case CREATE ASSERTION is not available but the system does support subqueries in table constraints. Notice how a table constraint avoids the need for the double negation that usually arises with tests for emptiness—instead of checking for the non-existence of a row that fails to satisfy a given condition, we give the inverse condition that every row must satisfy. For the sake of variety, Example 6.3 uses an invocation of EXCEPT in place of Example 6.2’s use of NOT IN.

Example 6.3: Use of NOT EXISTS

```
CREATE ASSERTION Must_be_enrolled_to_take_exam_alternative1
CHECK (
    NOT EXISTS ( SELECT StudentId, CourseId
                 FROM EXAM_MARK
                 EXCEPT
                 SELECT StudentId, CourseId
                 FROM IS_ENROLLED_ON ) ) ;
```

Example 6.3a: Alternative formulation as a table constraint

```

ALTER TABLE EXAM_MARK
ADD CONSTRAINT Must_be_enrolled_to_take_exam_alternative2
    CHECK ( EXISTS ( SELECT StudentId, CourseId
                      FROM   IS_ENROLLED_ON
                      WHERE  StudentId = EXAM_MARK.StudentId
                          AND   CourseId  = EXAM_MARK.CourseId )
    ) ;

```

In Example 6.3a, note the use of the table name, EXAM_MARK, as a range variable to qualify references to columns of that table. As always, the condition given as the operand of CHECK is one that would be legal as a WHERE condition following a FROM clause specifying just the table to which the constraint applies (viz., FROM EXAM_MARK in the case at hand).

Now, if the SQL implementation doesn't allow subqueries to appear in table constraints and doesn't support CREATE ASSERTION, then none of the formulations in Examples 6.2, 6.3, and 6.3a will be available. Happily, this particular constraint can be expressed as a foreign key constraint, as we shall see later in Section 6.4, the subsection headed **Foreign Keys**.

Example 6.4 is a translation into SQL of the corresponding example in the theory book, which is included there merely to show that for any scalar comparison there is an alternative formulation using IS_EMPTY.

Example 6.4: MAX_ENROLMENTS expressed using an invocation of NOT EXISTS

```

CREATE ASSERTION MAX_ENROLMENTS_alternative1
CHECK (NOT EXISTS (SELECT *
                   FROM (VALUES (SELECT COUNT(*)
                                FROM IS_ENROLLED_ON)) AS V(N)
                   WHERE V.N > 20000 ) ) ;

```

Explanation 6.4

- **VALUES (SELECT COUNT(*) FROM IS_ENROLLED_ON)** denotes a table with just one column, unnamed, in whose single row the value of that column is the number of rows in the current value of `IS_ENROLLED_ON`. The `SELECT` expression is parenthesized to make it into a scalar subquery and given as the argument to an invocation of `VALUES`, which makes the number denoted by that scalar subquery into a one-row, one-column table. (Actually, it might be safer to place an extra pair of parentheses around the `SELECT` expression here. Although `VALUES 1` and `VALUES (1)` are equivalent, it might not be clear as to which role the single parentheses are taking: do they denote a scalar subquery, as I have assumed, or are they the optional ones surrounding a single table expression? If the latter, we would expect a syntax error.)
- **AS V(N)** defines the range variable `V` to refer to what in this case is just the single row of that table, and also assigns the name `N` to its only column.
- **WHERE V.N > 20000** operates on that one-row, one-column table to yield a table of heading `(N INTEGER)` that is empty if and only if the single row in that one-row, one-column table fails to satisfy the condition `N > 20000`. Thus, the result is empty only when the number of enrolments is in fact no greater than the maximum allowed.



Vi vokser i Norge
og har virksomhet
helt frem til 2050

Er du interessert i sommerjobb
eller fast stilling?

Se informasjon om sommerjobber på
www.bp.no

bp



Use of Table Comparisons

Table comparisons are described in Chapter 5, Section 5.9, where it is noted that although table expressions cannot be compared, we have `TABLE (t)` to convert a table expression *t* into a value expression of type `ROW (r) MULTISSET`, where *r* is the row type of *t*. However, the only operator in SQL for comparing two multisets is “=”, so SQL has no direct counterparts of the theory book’s Examples 6.5, 6.6, and 6.7, which use “ \subseteq ”, and nor does this chapter. Those examples are shown merely to demonstrate that every constraint that can be expressed as an invocation of `IS_EMPTY` can be formulated alternatively as an invocation of “ \subseteq ”. If SQL were to have a counterpart of that operator, it would presumably have to be an “is submultiset of” operator, where *m1* is a submultiset of *m2* if and only if each element of *m1* appears at least as many times in *m2* as it does in *m1*. But SQL doesn’t have such an operator.

Effects of NULL

Here’s an important distinction between expressions denoting tables and expressions denoting multisets of rows: a table expression cannot evaluate to NULL, whereas a multiset expression can. Moreover, although a row expression can evaluate to NULL—for instance, `CAST (NULL AS ROW (X INTEGER, Y INTEGER))` is a legal expression—NULL cannot appear as an element of the body of a table. Every element of the body of a table is indeed a row. However, if a table has a column whose declared type is a row type, then NULL might appear in place of a value for that column in some row of that table.

It follows from the foregoing discussion that although a multiset expression in general can evaluate to NULL, an invocation of `TABLE` will never do so (recall that, counterintuitively, `TABLE` operates on a table and returns a multiset of rows). Nor can NULL ever appear as an element of a multiset resulting from an invocation of `TABLE`. However, the comparison `TABLE (t1) = TABLE (t2)`, where *t1* and *t2* are equal in cardinality, evaluates to UNKNOWN whenever NULL appears at any level of nesting within either of *t1* or *t2*.

Use of Truth-Valued Aggregate Operators

Example 6.8 in the theory book is an awkward one using double negation, offered as motivation for the neater way of expressing such constraints subsequently shown in Example 6.9. Example 6.8 reads as demanding that no exam mark shall not be in the required range, whereas Example 6.9 reads, more naturally, as requiring that every mark shall be in that range. Here, the SQL translations illustrate SQL’s `BETWEEN` and `NOT BETWEEN` shorthands for in-range tests.

Example 6.8: Restricting exam marks to between 0 and 100

```
CREATE ASSERTION Marks_between_0_and_100
CHECK ( NOT EXISTS ( SELECT *
                     FROM EXAM_MARK
                     WHERE Mark NOT BETWEEN 0 AND 100 ) ) ;
```

As mentioned in Chapter 5, SQL has `EVERY` as its counterpart of **Tutorial D**'s aggregate operator `AND`.

Example 6.9: Restricting exam marks to between 0 and 100 using `EVERY`

```
CREATE ASSERTION Marks_between_0_and_100_using EVERY
CHECK ( ( SELECT EVERY ( Mark BETWEEN 0 AND 100 )
        FROM EXAM_MARK ) ) ;
```

$x \text{ BETWEEN } y \text{ AND } z$ is equivalent to $x \geq y \text{ AND } x \leq z$.

It follows from Example 6.9 that if the SQL standard's `CREATE ASSERTION` and type `BOOLEAN` are both supported, then use of `EVERY` provides an alternative method of testing a table for being empty. If tx is a table expression, then we have the scalar subquery `(SELECT EVERY (FALSE) FROM (tx) AS T)`. When the result of tx contains a row, that row clearly fails to satisfy the condition `FALSE` and so the result of the scalar subquery is `FALSE`; otherwise the table is empty and the result is `UNKNOWN`, in which case the constraint is deemed to be satisfied, as previously explained. The reason why the result is `UNKNOWN` instead of what it should correctly be, viz. `TRUE`, is explained in **Effects of NULL**.

Effects of NULL

Let $aggop(x)$ be an invocation of some aggregate operator $aggop$ in SQL, where x is an expression (usually an open expression) to be evaluated against each row of the table t determined by the context in which the invocation appears. Then $aggop$ considers only those rows that satisfy the condition $x \text{ IS NOT NULL}$. It follows that if $aggop$ is `EVERY` or `SOME` and x evaluates to `TRUE` or `FALSE` for at least one row of t , then the result is either `TRUE` or `FALSE`, never `UNKNOWN`. However, if x evaluates to `UNKNOWN` for every row of t (which is true in the particular case when t is empty), then SQL's other general rule kicks in, requiring the result to be `NULL`, which is equivalent to `UNKNOWN` when it appears in the place of a `BOOLEAN` value. That anomaly is to some extent compensated for, when `EVERY` is used in constraint declarations, by SQL's rule that a constraint is deemed to be satisfied when it evaluates to `UNKNOWN`. However, `(SELECT SOME (TRUE) FROM (tx) AS T)` is not reliable as an existence test because it evaluates to `UNKNOWN` if the result of tx is empty, when a constraint based on that condition would be deemed satisfied. That problem could be addressed by writing `COALESCE ((SELECT SOME (TRUE) FROM (tx) AS T), FALSE)` or, equivalently, `(SELECT SOME (TRUE) FROM (tx) AS T) IS TRUE` (see Chapter 3, Section 3.5 **Deriving Predicates from Predicates**, Figure 3.1a in the subsection headed *Other monadics*).

6.4 Useful Shorthands for Expressing Some Constraints

Section 6.4 in the theory book describes three special classes of constraint and shorthands that have been proposed for them, not all of which have been adopted in **Tutorial D**. The three special classes are tuple constraints, key constraints, and foreign key constraints. SQL has counterparts of all three, as different kinds of table constraints.

CHECK Constraints

A **CHECK** constraint is a table constraint defined using the key word **CHECK**, as already illustrated in several examples in this chapter. In particular, a **CHECK** constraint can be used to express a constraint such as the one shown in Example 6.10, referred to in the theory book as a tuple constraint (so one might call it a row constraint in SQL). This is clearly the way most SQL users would prefer to express such a constraint—in fact, it is the only way when Examples 6.8 and 6.9 are unavailable for want of support for subqueries in constraints.

Example 6.10: Shorthand for a row constraint

```
ALTER TABLE EXAM_MARK
ADD CONSTRAINT Mark_in_range
CHECK ( Mark BETWEEN 0 AND 100 ) ;
```



Actually, it's not *quite* the only way. The constraint can be included in the column definition for `Mark` in the `CREATE TABLE` statement for `EXAM_MARK`, as shown in Example 6.10a. A constraint declared as part of a column definition is called a *column constraint*.

Example 6.10a: Column constraints included in a column definition

```
Mark INTEGER NOT NULL CHECK ( Mark BETWEEN 0 AND 100 ),
```

Example 6.10a has two separately declared column constraints in the same column definition. Both are unnamed but could be named if desired. Constraints that are declared without names acquire implementation-dependent names that show up in the database catalog. As in **Tutorial D**, naming a constraint allows it to be dropped when no longer needed (SQL uses the same key word, `DROP`).

The first column constraint, `NOT NULL`, is short for `CHECK (Mark IS NOT NULL)`. One might conclude that `BETWEEN 0 AND 100` would be allowed as short for `CHECK (Mark BETWEEN 0 AND 100)`, but that would be a wrong conclusion. One might also wonder if `CHECK (Mark BETWEEN 0 AND 100)` could be included (perversely) in the definition of some column other than `Mark`. In fact it can. Moreover, a column constraint can reference other columns in the same table, in which case the choice as to which column definition to include it in becomes arbitrary and one might prefer to write it as a regular table constraint (at the expense of an extra key word and a comma).

Effect of `NULL`

Until SQL:1999, if a column was subject to a `NOT NULL` constraint, then every value v appearing in that column could be guaranteed to compare equal with itself and not equal to every value of its type other than itself. That guarantee does not hold with all the additional types that were added to SQL in SQL:1999. For example, if a column is defined on type `ROW (x INTEGER, y INTEGER)`, then a `NOT NULL` constraint will not prevent the value `ROW (CAST (NULL AS INTEGER), 42)` appearing in that column. A similar comment applies to user-defined structured types, where the value of a component of the structure being `NULL` does not confer “nullness”, so to speak, on the whole value (see Chapter 2, Section 2.10, **Types and Representations**, the subsection **Effect of `NULL`**).

Keys

We have already seen one way of declaring a key in SQL, in `CREATE TABLE` statements. For example, the one for `EXAM_MARK` in the introduction to Chapter 5 includes the table constraint `PRIMARY KEY (StudentId, CourseId)`. This is almost equivalent to **Tutorial D**'s `KEY { StudentId, CourseId }`, the exceptions being: (a) there is some significance to the order in which the column names are written, as explained in the following section on foreign keys, and (b), as the key words `PRIMARY KEY` suggest, no more than one primary key can be specified for the same base table.

A `PRIMARY KEY` specification carries an implicit `NOT NULL` constraint on each column of the specified key. When more than one key constraint is required, the key word `UNIQUE` must be used in place of `PRIMARY KEY` for all or all but one of them. A `UNIQUE` specification does *not* carry an implicit `NOT NULL` constraint on each column of the specified key (says the SQL standard, though I am aware of at least one SQL implementation where it does).

Whether declared using `PRIMARY KEY` or `UNIQUE`, at least one column must be specified. SQL has no direct counterpart of **Tutorial D**'s `KEY { }`.

When a key consists of just one column it may be expressed in shorthand as a column constraint. For example, in the `CREATE TABLE` statement for `COURSE`, the primary key could be specified by adding `PRIMARY KEY` to the column definition for `CourseId`.

SQL differs from **Tutorial D** in its support for keys in the following respects:

- SQL does not require at least one key for every base table. In **Tutorial D**, if no key is explicitly declared, then `KEY { ALL BUT }` is implicit.
- When no key is specified there is no prohibition on multiple appearances of the same row.
- SQL does not recognize the empty set as a key.
- SQL allows a key to be a proper superset of another key for the same base table. (This “feature” is sometimes used as a workaround for the fact that the columns of the foreign key are required to correspond to those of a declared key of the referenced table.)

Effects of `NULL`

When a `UNIQUE` specification u for base table t includes a column c that is not subject to a `NOT NULL` constraint, the appearance of several rows having `NULL` in place of a value for c and equal values for the other columns specified in u is permitted. It is only when each column of the specified “key” has a value that those column values may not appear in the same combination in more than one row of t .


WHEN/THEN Key Constraints

Temporal databases are beyond the scope of the theory book, but the problems that arise with them and proposed solutions to those problems are described in detail in reference [12], which presents its proposals as notional extensions to **Tutorial D**. One of these extensions is a special shorthand called a `WHEN/THEN` constraint, and SQL has a somewhat similar solution to the particular problem addressed by such constraints (though it falls far short of addressing all of the problems described in reference [12]).

Suppose a table has two columns representing a period of time throughout which the information conveyed by the other columns is recorded as having been the case. A salary history table for employees, with columns `From` and `To` for dates defining the applicable time periods, would be a good example. A constraint is needed to avoid the possibility of an employee being shown as having two different salaries on the same day, which could happen if two rows for the same employee have overlapping periods indicated by their `From` and `To` dates. The term “WHEN/THEN constraint” appeals to the notion of “unpacking” the table so that each row is replaced by one or more rows, one for each date contained in its from-to period: *when* the relation is unpacked, *then* the given key constraint (e.g., on employee number and date) is to hold.

Here’s a concrete example showing how SQL supports WHEN/THEN constraints.

```
CREATE TABLE SAL_HISTORY ( EmpNo CHAR(6),  
                             Salary INTEGER NOT NULL,  
                             From DATE  
                             To DATE  
                             PERIOD FOR During ( From, To ),  
                             PRIMARY KEY ( EmpNo, During WITHOUT OVERLAPS )  
                             ) ;
```



The advertisement for Gaieteye features a background image of a person running on a path during a sunrise or sunset. The Gaieteye logo, consisting of a yellow square with a stylized 'G' and the word 'gaieteye' in white, is in the top left. Below it is the tagline 'Challenge the way we run'. In the center-left, the text 'EXPERIENCE THE POWER OF FULL ENGAGEMENT...' is displayed above a horizontal dotted line. Below this, the text 'RUN FASTER. RUN LONGER.. RUN EASIER...' is shown. On the right side, there is a yellow button with the text 'READ MORE & PRE-ORDER TODAY' and 'WWW.GAITEYE.COM'. A hand cursor icon is pointing at the button. A white line with a circular end points from the text 'RUN EASIER...' towards the runner's feet.



The `PERIOD FOR` specification states that the `From` and `To` values in each row denote a time interval (called a period because SQL uses the term “interval” for something else). The `From` values are treated as closed bounds, the `To` values as open bounds, so a given row in `SAL_HISTORY` indicates that an employee was paid a certain salary from the given `From` date up to *but not including* the given `To` date. The specification implies the column constraint `NOT NULL NOT DEFERRABLE ENFORCED` for each of columns `From` and `To`.

During `WITHOUT OVERLAPS`, which, if required, must appear as the last element of the key, specifies that if the same `EmpNo` value appears in two distinct rows of `SAL_HISTORY`, then the `From` and `To` values in those rows must denote `During` periods that do not overlap (have no date in common).

Historical Notes and Comments

Support for keys (and foreign keys) arrived in 1989, as part of an addendum to SQL:1987, the first international edition of the SQL standard.

`PERIOD FOR` and `WITHOUT OVERLAPS` arrived as an optional conformance feature in SQL:2011. Note that although the `WITHOUT OVERLAPS` specification in `SAL_HISTORY` prevents an employee from being recorded as having two or more different salaries on the same day, it does not enforce the “packed form” defined in reference [12]. In the given example, packed form would prevent the appearance of two or more rows with consecutive `From-To` periods showing the same salary for the same employee—a case of what reference [12] calls *circumlocution*. Clearly, two such rows can be replaced by a single row having the `From` value of the earlier period and the `To` value of the later one.

A question arises as to what happens if one of those implied `NOT NULL` constraints is dropped or altered to be `NOT ENFORCED` (the historical note in Section 6.1 shows how this might be done in accordance with the SQL standard). SQL:2011 is silent on that possibility.

Foreign Keys

(See the corresponding section in the theory book for the meaning of this term.)

Examples 6.2, 6.3, and 6.3a are alternative ways of formulating a constraint that enforces a business rule to the effect that every student who takes an exam must be enrolled on the applicable course. As it happens, that constraint can also be formulated as a foreign key, expressed as a table constraint for base table `EXAM_MARK`.

Example 6.3b: Alternative formulation for 6.3 as a foreign key constraint

```
ALTER TABLE EXAM_MARK
ADD CONSTRAINT Must_be_enrolled_to_take_exam_alternative3
FOREIGN KEY ( StudentId, CourseId )
REFERENCES IS_ENROLLED_ON ;
```

The formulation in Example 6.3b is available only because the following conditions hold:

1. There is a one-to-one correspondence from the specified columns, in the specified order, to those of the primary key of `IS_ENROLLED_ON`. Corresponding columns do not have to have the same name but they must be of the same declared type. The table name for the *referenced table* (`IS_ENROLLED_ON` in the example) can be followed by a commalist of column names in parentheses, in which case that commalist—the *referenced columns*—must correspond exactly, in the correct order, to some key specified for the referenced table. The referenced columns must be explicitly specified when the applicable key is declared using `UNIQUE` rather than `PRIMARY KEY`, or when it is declared using `WITHOUT OVERLAPS`.
2. The referenced table and the referencing table (`EXAM_MARK` in the example) are both base tables.

A foreign key declaration in SQL can include a specification of a *compensatory action*, which defines an additional update to take place automatically when the constraint would otherwise be violated. For example, the specification `ON DELETE CASCADE`, when added to the foreign key declaration in Example 6.3b, states that when a row is deleted from the referenced table, `IS_ENROLLED_ON`, all matching rows in `EXAM_MARK` are to be deleted too. Similarly, `ON UPDATE CASCADE` specifies that when the `StudentId` or `CourseId` value of some row in `IS_ENROLLED_ON` is updated, the new value is propagated to all of the matching rows in `EXAM_MARK`. For another example, `ON DELETE SET DEFAULT` specifies that when a row in `IS_ENROLLED_ON` is deleted, the values for columns `StudentId` and `CourseId` in the matching rows of `EXAM_MARK` are replaced by the default values for those columns—in which case those default values must be sure to be matched by some row in the referenced table, of course.

A compensatory action, being a further update of some kind, might in turn result in violation of a foreign key constraint that might in turn have a compensatory action defined for it. The interactions between compensatory actions and triggered procedures are fully specified in the SQL standard but can be bewilderingly complicated.

When no compensatory action is required, SQL has two ways of dealing with foreign key constraint violations and allows the user to choose between the two. The two options are `NO ACTION` and `RESTRICT`. `ON DELETE NO ACTION` and `ON UPDATE NO ACTION` are self-explanatory: a constraint violation is to cause the delete or update to be rejected. But `RESTRICT`, rather than `NO ACTION` is the default option. `ON DELETE RESTRICT` and `ON UPDATE RESTRICT` can cause a delete or update to be rejected even before the overall effect of the statement has been evaluated and even when the overall effect would be accepted under `NO ACTION`. For example, suppose the table `T` is subject to the constraint

```
FOREIGN KEY (FK) REFERENCES T(K) ON UPDATE RESTRICT
```

and the following statement is executed:

```
UPDATE T SET K = K + 1 ;
```

If `T` has a row with 3 for column `K` and one or more rows with 3 for `FK`, the update is rejected even if `T` also has a row with 2 for `K` that will satisfy the foreign key constraint when 3 replaces 2. `ON UPDATE NO ACTION` had been specified instead, the update would be accepted because the overall effect would not cause a constraint violation—the constraint is properly checked at the statement boundary instead of being checked against some intermediate state that arises mid-execution.



Strømmen produseres ofte langt fra der den skal brukes.

Statnett sitt oppdrag er å gjøre strømmen tilgjengelig, uansett hvor i dette langstrakte landet du bor. Det er vi som bygger og drifter "riksveiene" i norsk strømforsyning. Gjennom vårt landsdekkende nett sørger vi for en sikker fordeling av strøm mellom nord, sør, øst og vest.

Vi binder Norge sammen

Statnett
Vårt felles kraftnett

Er du student? Les mer her
www.statnett.no/no/Jobb-og-karriere/Studenter



If the applicable key of the referenced table is defined using a period name and `WITHOUT OVERLAPS`, then the definition of the referencing table must include a period name *pn* defined on columns compatible with the corresponding ones of the referenced table, and the foreign key declaration must include `PERIOD pn`. The foreign key constraint is then considered to apply to the unpacked forms of the referencing and referenced tables, and the `ON DELETE/ON UPDATE` options are not supported—`RESTRICT` is implicit.

6.5 Updating Tables

Section 6.5 in the theory book is headed **Updating Relvars**. I could perhaps have used the heading **Updating Table Variables** here but such terminology is not used in SQL. Nevertheless, of course it is table variables—base tables or updatable views—that are updated, not tables *per se*. As the theory book does not cover the difficult and somewhat controversial topic of updating virtual relvars (as views are called in that book), this book likewise considers only base tables as targets. Also omitted, as in the theory book, is any discussion of SQL's comprehensive provisions for security and authorization, giving control over (among many other things) which users are authorized to do what kinds of updating to which tables.

The theory book introduces the topic of updating by describing the assignment operator, “:=” in **Tutorial D**. SQL uses a different syntax for assignment, using the key word `SET` and “=”. Thus, to add 1, so to speak, to the integer variable *x*, SQL has `SET x = x + 1`. However, the operator is not supported at all for tables, so SQL has no direct counterpart of the theory book's Example 6.11. It does, however, have counterparts of **Tutorial D**'s `INSERT`, `UPDATE`, and `DELETE` operators, which we can deal with here quite briefly by giving translations to SQL of the theory book's examples 6.12, 6.13, 6.14, and 6.16. (Example 6.15 is missing because that one uses “:=”.)

INSERT

Loosely speaking, `INSERT` takes the rows of a given *source* table and adds them to the specified *target* table, retaining all the existing rows in the target. Example 6.12 shows how `INSERT` can be used to add a single row to `IS_ENROLLED_ON`.

Example 6.12: Enrolling a student on a course using `INSERT`

```
INSERT INTO IS_ENROLLED_ON VALUES ('S3', 'C2') ;
```

Recall that `VALUES ('S3', 'C2')` denotes the table consisting of just the row `('S3', 'C2')`. If that row already exists in the target table, then the update has the effect of increasing the number of appearances of that row by one, unless some key is specified for that table (as is the case with `IS_ENROLLED_ON`), in which case the update fails.

Recall also that the columns of the result of a `VALUES` expression are effectively unnamed, so the column ordering has to be used to determine the correspondence between source and target columns. In Example 6.12, 'S3' becomes the `StudentId` value in the inserted row and 'C2' becomes the `CourseId` value, and that's because `StudentId` is defined to be the first column of `IS_ENROLLED_ON`, `CourseId` the second. However, the defined ordering can be explicitly overridden, as shown in Example 6.12a.

Example 6.12a: Overriding the defined column ordering

```
INSERT INTO IS_ENROLLED_ON (CourseId, StudentId)
VALUES ('C2', 'S3') ;
```

A `VALUES` expression is not restricted to tables of just one row. For example, the source table `VALUES ('S3', 'C2'), ('S4', 'C1')` would simultaneously enroll student S3 on course C2 and S4 on C1. In general, any table expression can be used as the source table for an `INSERT` invocation, just as any relational expression can be used for the same purpose in **Tutorial D**.

Example 6.13, like its counterpart in the theory book, illustrates the convenience of allowing any table expression to be the source for an `INSERT`. It assumes that all the exam scripts submitted by students have been marked and it has been decided to record marks of zero for students who failed to turn up for an exam they should have sat. (Remember that SQL's `EXCEPT` requires its operands to be of the same degree, unlike **Tutorial D**'s `NOT MATCHING`—hence the third element, 0, of the second operand in the example.)

Example 6.13: Awarding zero marks to students who failed to take the exam

```
INSERT INTO EXAM_MARK
SELECT StudentId, CourseId, 0
FROM IS_ENROLLED_ON
EXCEPT
SELECT StudentId, CourseId, 0
FROM EXAM_MARK ;
```

UPDATE

Loosely speaking, `UPDATE` changes some of the column values of some existing rows of its target table. Thus, although some rows disappear from the target and others arrive in it, so to speak, the cardinality of the table does not change. Suppose the exam board for course C2 decides that the exam has been marked too harshly and everybody's mark is to be increased by 5. Example 6.14 shows how.

Example 6.14: Adding 5 to all the marks for course C2

```
UPDATE EXAM_MARK SET Mark = Mark + 5
WHERE CourseId = 'C2' ;
```

The syntax is self-explanatory. The `WHERE` specification is optional and defaults to `WHERE TRUE`, meaning that the specified changes are to be applied to all existing rows in the target table. The expression `Mark = Mark + 5` is a *column assignment*. When several column assignments are needed they are separated by commas and the semantics of multiple assignment as described in the theory book apply: the right-hand sides are all evaluated before any column assignments are performed. The same column cannot be the target or more than one assignment.

SQL's `UPDATE` differs from **Tutorial D's** in the following interesting respect. Let relvar rv be assigned the relation `RELATION { TUPLE { X 1, Y 2 }, TUPLE { X 2, Y 2 } }`. Then `UPDATE rv WHERE X = 1 (X := 2)` causes rv to consist of just a single tuple, `TUPLE { X 2, Y 2 }`. The SQL counterpart, assuming no constraint violation would arise, causes the target table to contain two appearances of the row `ROW (2, 2)`.

Hva får egentlig en ingeniør- eller teknologistudent for 300 kroner?

- Medlemskap i en aktiv studentorganisasjon – hele studietiden
- 150 tillitsvalgte studenter som ivaretar dine interesser
- Jobbsøkerkurs
- Gratis PC-forsikring og gode bank- og forsikringstilbud
- Teknisk Ukeblad og NITO Refleks
- Møteplasser på web 2.0

Flere medlemsfordeler og innmelding: www.nito.no/student

Alle som studerer på ingeniør-, bioingeniør-, sivilingeniør eller andre teknologistudier (høgskolekandidat, bachelor eller master) kan bli medlem i NITO.

NITO NORGES STØRSTE ORGANISASJON
FOR INGENIØRER OG TEKNOLOGER



If the target table has a period name defined on two of its columns, then an UPDATE statement for that table can include a FOR PORTION OF clause, specifying a FROM and a TO value. In this case the cardinality of the target table *can* change. For example, if the SAL_HISTORY table contains the following rows:

```
('123456', 55000, DATE('2011-09-01'), DATE('2012-08-01')),
('123456', 60000, DATE('2012-08-01'), DATE('9999-12-31'))
```

and it is discovered that employee 123456's salary was in fact increased to 60000 on July 1st, 2012, then Example 6.14a can be used to make the necessary correction. As a result, those two rows will be replaced by the following three:

```
('123456', 55000, DATE('2011-09-01'), DATE('2012-07-01')),
('123456', 60000, DATE('2012-07-01'), DATE('2012-08-01')),
('123456', 60000, DATE('2012-08-01'), DATE('9999-12-31'))
```

Note that the second and third exhibit circumlocution: using more than one row to state what could equivalently be stated by a single row showing that employee 123456's salary is 60000 from July 1st, 2012 until SQL's rather pessimistic estimate of the end of time (this being what is sometimes used to indicate “indefinitely”).

Example 6.14a: Updating a “portion” of the salary history table

```
UPDATE SAL_HISTORY
FOR PORTION OF During
                FROM DATE('2012-07-01') TO DATE('2012-08-01')
SET ( Salary = 60000 )
WHERE EmpNo = '123456' ;
```

DELETE

Loosely speaking, DELETE removes some existing rows from its target table. Suppose the university decides that course C3 is to be withdrawn. Example 6.16 shows how.

Example 6.16: Withdrawing course C3, using DELETE

```
DELETE FROM COURSE WHERE CourseId = 'C3' ;
```

Every row that satisfies the given WHERE condition is deleted; rows that do not satisfy it remain in place.

As with UPDATE, a FOR PORTION OF clause can be specified if the target table has a defined period name, as illustrated in Example 6.16a.

Example 6.16a: Deleting a “portion” of the salary history table

```
DELETE SAL_HISTORY
FOR PORTION OF During
                FROM DATE('2012-01-01') TO DATE('2012-02-01')
WHERE EmpNo = '123456' ;
```

As a result, the row

```
('123456', 55000, DATE('2011-09-01'), DATE('2012-08-01'))
```

is replaced by the two rows

```
('123456', 55000, DATE('2011-09-01'), DATE('2012-01-01')),
('123456', 55000, DATE('2012-02-01'), DATE('2012-08-01'))
```

and the DELETE statement will have effected an increase in cardinality instead of the usual decrease.

MERGE and TRUNCATE

SQL has two more table update operators, MERGE and TRUNCATE.

MERGE, like INSERT, takes a source table *s* and uses it to update a target table *t*. Briefly, a MERGE statement specifies a matching condition to determine which rows of *s* have at least one matching row in *t* (under that specified matching condition). It then specifies an open-ended series of conditions to be applied to each row of *s* paired with actions to be applied on *t*. WHEN MATCHED AND *c1* THEN *x1* specifies that action *x1*, necessarily an UPDATE or DELETE, is to be applied on *t* for each matching row in *s* that satisfies the condition *c1*. WHEN NOT MATCHED AND *c2* THEN *x2* specifies that action *x2*, necessarily an INSERT, is to be applied on *t* for each non-matching row in *s* that satisfies the condition *c2*.

The curiously named TRUNCATE statement deletes all the rows from its specified target, bypassing any triggered actions, including compensatory actions, specified for that target. The target must be a base table.

Multiple Assignment

SQL supports multiple assignment to local variables and also applies multiple assignment semantics in SET clauses of UPDATE statements, but does not support multiple assignment in connection with updates on table targets. Thus, SQL has no counterpart to the theory book's Example 6.17, simultaneously deleting from both COURSE and IS_ENROLLED_ON. If we assume that there must be at least one enrolment for each course, and that students can enroll only on existing courses, deferred constraint checking has to be used, as shown in Example 6.17 here.

Download free eBooks at bookboon.com

Example 6.17: Withdrawing course C3 and deleting any enrolments on C3

Assume the definition of IS_ENROLLED_ON includes

```
CONSTRAINT Course_must_exist_for_enrolment
FOREIGN KEY ( CourseId ) REFERENCES COURSE ON DELETE NO ACTION
and the definition of COURSE includes
```

```
CONSTRAINT Enrolment_must_exist_for_course
CHECK ( CourseId IN ( SELECT CourseId FROM IS_ENROLLED_ON )
```

Then the desired effect can be achieved by this:

```
SET CONSTRAINTS Course_must_exist_for_enrolment DEFERRED ;
DELETE FROM COURSE WHERE CourseId = 'C3' ;
DELETE FROM IS_ENROLLED_ON WHERE CourseId = 'C3' ;
SET CONSTRAINTS Course_must_exist_for_enrolment IMMEDIATE ;
```

ON DELETE NO ACTION states that no compensatory action is to be used for enforcement of the foreign key constraint. Deferring the checking of that constraint allows the first DELETE statement to succeed in spite of the consequent existence of “orphan” rows in IS_ENROLLED_ON. Cancelling the deferment immediately after the second delete then causes the constraint to be checked. If the DELETE statements were the other way around, then we would have to defer Enrolment_must_exist_for_course instead.





Skatteetaten



Vil du jobbe i et av landets største IT-miljøer?

Vi skal gjøre det kompliserte enkelt

Skatteetaten tilbyr store fagmiljø og utfordrende oppgaver innen:

- > Systemutvikling
- > Service oriented architecture (SOA)
- > Business intelligence (BI)
- > Testledelse
- > Webutvikling
- > IT sikkerhet
- > Infrastruktur
- > Brukergrensesnitt

For nyutdannede IT-spesialister kan vi tilby et to-årig traineeprogram.

Profesjonell • Nytenkende • Imøtekommende

For mer informasjon se skatteetaten.no/jobb



In Example 6.18, a straight translation of its counterpart in the theory book, the second statement, assigning to both X and Y, illustrates multiple assignment to local variables in SQL.

Example 6.18: A consequence of simultaneity

```
SET X = 1;
SET ( X, Y ) = ROW ( X + 1, X + 1 );
```

As in **Tutorial D**, the value 2 is assigned to both X and Y. The simultaneous assignment to X and Y can also be expressed using a `SELECT ... INTO` statement, as shown in Example 6.18a. An `INTO` clause can be used only in the first `SELECT` clause of such a statement and only when the resulting table contains no more than one row. (When a `SELECT` expression contains further `SELECT` expressions, the first `SELECT` clause is the one belonging to the outermost `SELECT` expression. The outermost `SELECT` expression cannot be combined with another by `UNION`, `EXCEPT`, or `INTERSECT`.)

Example 6.18a: Multiple assignment using `SELECT ... INTO`

```
SELECT * INTO X, Y
FROM   VALUES ( X + 1, X + 1 ) AS T;
```

The key word `ROW` in Example 6.18 is optional. That being the case, you might ponder the distinctions among the statements listed in Example 6.18b. Which ones assign 1 to X and which assign `ROW (1)`? Is (b) legal if the declared type of X is `ROW (F1 INTEGER)`. Is (e) legal if the declared type of X is `INTEGER`?

Example 6.18b: Some puzzling syntactic variations

- a) `SET (X) = (1);`
- b) `SET X = 1;`
- c) `SET (X) = 1`
- d) `SET (X) = ((1));`
- e) `SET X = (1);`

Effects of NULL

If the row expression given as the source for a multiple assignment evaluates to `NULL`, then `NULL` is assigned to each target.

If a `SELECT ... INTO` statement results in an empty table, then the target variables are not updated and a completion condition is given to indicate that. This is not exactly an “effect of `NULL`”, of course, but I mention it here because of the contrast with a row subquery, which delivers a single row with `NULL` for each field when its table expression evaluates to an empty table. This observation might influence the choice between `SET` and `SELECT ... INTO`.

Historical Notes

`INSERT`, `UPDATE`, and `DELETE` have been in SQL from the beginning. `MERGE` was added in SQL:2003, `TRUNCATE` in SQL:2007.

Support for local variables and various programming language constructs is defined in Part 4 of the SQL standard, referred to as SQL/PSM. Part 4 first appeared in 1996, as an addendum to SQL:1992.

Support for multiple assignment to local variables was added in SQL:2003. `SELECT ... INTO` has been in SQL from the beginning, though until 1996 it was available only with “host” variables as targets, a host variable being one declared using some language other than SQL in a program written in that language. References to host variables are distinguished from references to SQL variables by prefixing them with colons (e.g., `:X`).

Transactions

For **Tutorial D**’s `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK`, SQL has the same syntax except for `START` in place of `BEGIN`. However, `START TRANSACTION` is used only for outermost transactions and cannot be given when a transaction has been started and not completed. Inner transactions are started using a `SAVEPOINT` statement, giving a name—a savepoint name—that identifies the database state at the time of execution. If `SAVEPOINT SN1` has been given, for example, then `RELEASE SAVEPOINT SN1` has the same effect as a **Tutorial D** `COMMIT` for all updates performed since savepoint `SN1` was established—it merely relinquishes the possibility of cancelling just those updates and does not make their effects visible to other users. To cancel those updates `ROLLBACK TO SAVEPOINT SN1` is given, but then the savepoint name `SN1` remains in existence. In both cases, any further existing savepoints, established after `SN1`, are destroyed.

If an attempt is made to update the database when no transaction has been explicitly started, then a transaction is implicitly started. When no transaction has been started, a `SET TRANSACTION` statement can be given to specify various options to override the defaults that otherwise apply to the next transaction. The options can alternatively be specified in a `START TRANSACTION` statement. The options in effect will apply when a transaction is implicitly started or when it is started by a `START TRANSACTION` statement that does not override them.

One of the options for `SET/START TRANSACTION` is the so-called *isolation level*, which applies to the whole of the outermost transaction. The default isolation level is `SERIALIZABLE`, this being the only one that enforces all of the normally defined properties of transactions. The weakest level, `READ UNCOMMITTED` allows other concurrent users to see the effects of updates that have not yet been committed (and might never be, of course). Intermediate levels, `READ COMMITTED` and `REPEATABLE READ`, as well as `UNCOMMITTED`, allow a transaction to perceive changes to the database that have been effected by other, committed transactions (for example, by evaluating the same table expression more than once, without updating the database between times, and getting different results).

Historical Notes

`SET TRANSACTION` appeared in `SQL:1992`. `START TRANSACTION` and `SAVEPOINT` were added in `SQL:1999`.



OLJE- OG ENERGIDEPARTEMENTET



Er du full av energi?

Olje- og energidepartementets hovedoppgave er å tilrettelægge for en samordnet og helhetlig energipolitikk. Vårt overordnede mål er å sikre høy verdiskapning gjennom effektiv og miljøvennlig forvaltning av energiresursene.

Vi vet at den viktigste kilden til læring etter studiene er arbeidssituasjonen. Hos oss får du:

- Innsikt i olje- og energisektoren og dens økende betydning for norsk økonomi
- Utforme fremtidens energipolitikk
- Se det politiske systemet fra innsiden
- Høy kompetanse på et saksfelt, men også et unikt overblikk over den generelle samfunnsutviklingen
- Raskt ansvar for store og utfordrende oppgaver
- Mulighet til å arbeide med internasjonale spørsmål i en næring der Norge er en betydelig aktør

Vi rekrutterer sivil- og samfunnsøkonomer, jurister og samfunnsvitere fra universiteter og høyskoler.

www.regjeringen.no/oed



Se ledige stillinger her

www.jobb.dep.no/oed



EXERCISES

1. SQL does not allow the empty set to be specified using `PRIMARY KEY` or `UNIQUE`. Write a table constraint that could be included in the definition of table `T` to simulate **Tutorial D**'s `KEY { }`. Mention any optional conformance features of SQL that your solution uses.

2. Using some SQL implementation that is available to you, try out Example 6.1b on it, using a low number, say 1, in place of 20000. Is it accepted? If so, does it have the intended effect? If not, is it accepted when you place `BEGIN` and `END` around the `RETURN` statement? If it's still not accepted, try `RETURNS INTEGER` instead of `RETURNS BOOLEAN`, and move the comparison from the function body to the constraint. Now is it accepted? And if so, does it have the desired effect? Finally, try specifying a different base table, say `IS_CALLED`, in the `ALTER TABLE` statement (and drop the constraint from `IS_ENROLLED_ON`). When `IS_CALLED` is nonempty and `IS_ENROLLED_ON` is at its maximum cardinality, is `INSERT INTO IS_ENROLLED_ON ...` accepted? Or does the DBMS check the constraint only on updates to `IS_CALLED`?

3. Suppose the table definition for `COURSE` is extended to include a column `MaxExamMark`, whose value in each row is the maximum mark obtainable for that course's exam. `{StudentId, CourseId}` is a foreign key in `EXAM_MARK`, referencing `IS_ENROLLED_ON`. A constraint is needed to ensure that no student is awarded a mark greater than the relevant maximum.

- a) Write an SQL `ALTER TABLE` statement to address this requirement.
- b) Complete the following statement to make it equivalent to your solution for part (a):

```
CREATE ASSERTION ...
    CHECK ( SELECT EVERY ( ... ) FROM EXAM_MARK ) ;
```

4. Now suppose that instead of there being a recorded maximum mark of each exam the maximum score for each question in each exam is recorded in the following relvar:

```
CREATE TABLE EXAM_QUESTION
( CourseId CID,
  Question# INTEGER,
  MaxMark INTEGER,
  PRIMARY KEY ( CourseId, Question# ) ;
```

For each course, the exam questions are supposed to be numbered sequentially, starting at 1.

- a) Write an SQL `CREATE ASSERTION` statement to address this requirement.
- b) Suppose the questions are subdivided into parts, a, b, c and so on, up to a maximum of six parts, and maximum marks are given for each part rather than for each question. Again, the parts for each question must be “numbered” sequentially, starting at a. Write an SQL `CREATE ASSERTION` statement to address *this* requirement.
- c) Devise shorthands, in the style of SQL, for expressing constraints of the kinds found in your solutions to a. and b.

5. Using the suppliers-and-parts database shown in Figure 4.13, define SQL integrity constraints to express the following requirements:

- a) Every shipment row must have a supplier number matching that of some supplier row.
- b) Every shipment row must have a part number matching that of some part row.
- c) All London suppliers must have status 20.
- d) No two suppliers can be located in the same city.
- e) At most one supplier can be located in Athens at any one time.
- f) There must exist at least one London supplier.
- g) The average supplier status must be at least 10.
- h) Every London supplier must be capable of supplying part P2.

6. For each example in Exercise 5, list the different kinds of update operation that, if permitted, would cause the constraint to be violated.

7. A database contains base tables T1 and T2. At all times at least one of these must be empty. The SQL implementation does not support `CREATE ASSERTION` but does allow subqueries to appear in table constraints. How can the stated requirement be implemented?

8. (Repeated from the body of the chapter.) Ponder the distinctions among the following examples.

- a) `SET (X) = (1) ;`
- b) `SET X = 1 ;`
- c) `SET (X) = 1`
- d) `SET (X) = ((1)) ;`
- e) `SET X = (1) ;`

Download free eBooks at bookboon.com

Which ones assign 1 to X and which assign ROW (1)? Is (b) legal if the declared type of X is ROW (F1 INTEGER). Is (e) legal if the declared type of X is INTEGER? You might like to try these out in some SQL implementation.

9. SQL has two ways of starting a transaction, `START TRANSACTION` for an outermost transaction and `SAVEPOINT` for inner ones. Describe any advantages and disadvantages you can think of for this scheme over one that uses the same method for all transactions.

10. SQL's `UNION`, `EXCEPT`, and `INTERSECT` operators are the only ones that have a `CORRESPONDING` option to specify that columns of two tables are to be paired by their names rather than their ordinal positions. List as many other operators and syntactic constructs in SQL that you can think of to which a `CORRESPONDING` option might usefully be added.

11. Consider the SQL implementation you are most familiar with. To what extent does it correctly support the standard features mentioned in this book? Is it relationally complete?



HELT GRATIS!

S for Skikk & Bank

En bok om ting som er greit å vite når du har flyttet hjemmefra.

dnb.no

DNB

Bank fra A til Å



Appendix A: References and Bibliography

Apart from [9], a closely related work that I recommend, these are all referenced in the body of the book.

- [1] Frederick P. Brooks Jr.: *The Mythical Man-Month* (20th anniversary edition). Reading, Mass.: Addison-Wesley (1995)
- [2] Stephen Cannan and Gerard Otten: *SQL—The Standard Handbook*. Maidenhead, UK: McGraw-Hill International (UK) Limited (1993).

This book covers SQL:1992. Stephen Cannan was an active participant in ISO/IEC JTC 1/SC 32/WG 3, the working group responsible for the development of the standard and he later became the convenor (ISO's term for chairperson) of that working group.

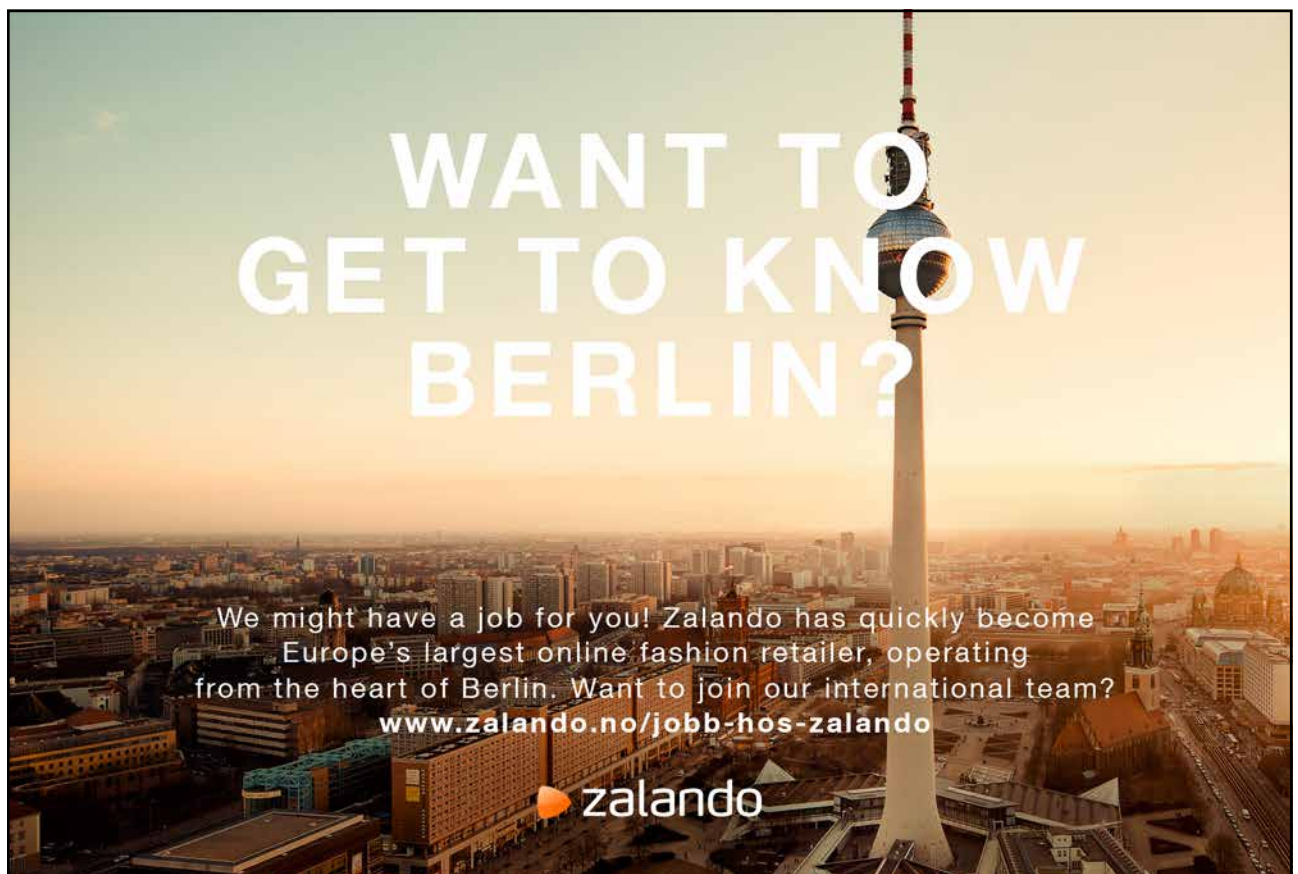
- [3] E.F. Codd. "A Relational Model of Data for Large Shared Data Banks", *CACM* 13, no. 6 (June 1970). Republished in "Milestones of Research", *CACM* 25, No. 1 (January 1982).
- [4] E.F. Codd: "A Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif. (November 1971).
- [5] E.F. Codd: *The Relational Model for Database Management—Version 2*. Reading, Mass.: Addison-Wesley (1990).
- [6] E.F. Codd: "Extending the Database Relational Model to Capture More Meaning", *ACM TODS* 4, No. 4 (December 1979).
- [7] Hugh Darwen: *An Introduction to Relational Database Theory*. A free download available, like the present book, at <http://bookboon.com/en/textbooks/it-programming>
- [8] Hugh Darwen: *Business System 12*. (PDF, slides and notes), available at www.northumbria.ac.uk/sd/academic/ceis/enterprise/third_manifesto/
- [9] C.J. Date: *SQL and Relational Theory: How to Write Accurate SQL Code* (2nd edition). Sebastopol, Calif.: O'Reilly (2012).

Although Date's book differs considerably from the present one in aim, organization, and target audience, it also uses **Tutorial D** for examples showing comparison of SQL with relational theory. It differs in length, too: 428 pages. There are exercises, many suitable for readers of the present book, at the end of each chapter—answers are given in an appendix.

- [10] C.J. Date and Hugh Darwen: *A Guide to the SQL Standard* (4th edition). Reading, Mass.: Addison-Wesley (1997)

The 4th and last edition of this book covers SQL:1996, whose main new features were SQL/PSM (the programming language) and SQL/CLI (“call level interface”, as in Microsoft's ODBC).

- [11] C.J. Date and Hugh Darwen: *Database Explorations: Essays on The Third Manifesto and Related Topics*. Trafford Publishing (2010).
- [12] C.J. Date, Hugh Darwen, and Nikos A. Lorentzos: *Temporal Data and The Relational Model*. San Francisco, Calif: Morgan Kaufmann (2003).



- [13] Lex de Haan and Toon Koppelaars: *Applied Mathematics for Database Professionals*. Berkeley, Ca.: Apress (2007).

See reference [13] in the theory book, Appendix B.

- [14] Patrick A.V. Hall; P. Hitchcock, Stephen Todd (January 1975). “An algebra of relations for machine computation”. *Conference record of the second ACM Symposium on the Principles of Programming Languages*. Palo Alto, California: ACM. pp. 225–232
- [15] International Organization for Standardization (ISO): *Information Technology—Database Languages—SQL*, Document ISO/IEC 9075:2011 (2011).
http://www.iso.org/iso/home/store/catalogue_tc/catalogue_tc_browse.htm?commid=45342
- [16] Jim Melton and Alan R. Simon: *Understanding The New SQL: A Complete Guide*. San Mateo, Calif.: Morgan-Kaufman Publishers (1993)

Another guide to SQL: 1992. Jim Melton was, and at the time of writing (2012) still is, the editor of the SQL standard.